OBJECT-ORIENTED DESIGN AND IMPLEMENTATION OF A PARALLEL ADA SIMULATION SYSTEM

THESIS

James T. Belford
Captain, USAF

AFIT/GCE/ENG/93D-01

AFIT/GCE/ENG/93D-01

OBJECT-ORIENTED DESIGN AND IMPLEMENTATION OF A PARALLEL ADA SIMULATION SYSTEM

THESIS

Presented to the faculty of the Graduate School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

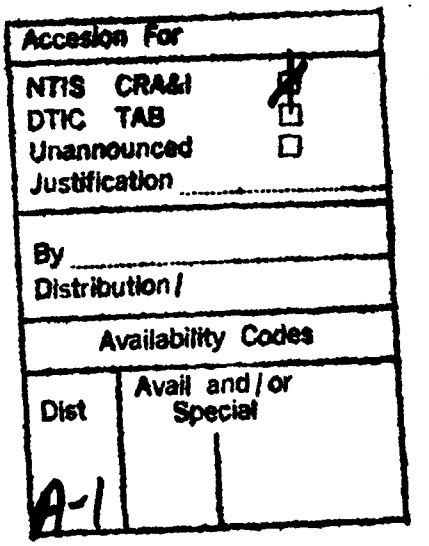
Requirements for the Degree of

Master of Science in Computer Engineering

James T. Belford

Captain, USAF

November, 1993



Approved for public release; distribution unlimited

Acknowledgments

First, I would like to thank my thesis advisor, Dr. Thomas Hartrum, and committee members, Lt Col Pat Lawlis and Maj Paul Bailor for their unwavering assistance throughout this research effort. A special thanks is in order for Maj Eric Christensen, USA. Without Maj Christensen's support and guidance, successful completion of this research effort would not have been possible.

Next, I would like to thank my family. My wife Maribel was very helpful when it came to keeping my sanity. She accepted more then her share of responsibility in the area of running our household, which allowed me to apply more of my time to school work. Her positive words when things became tough provided the strength required to persevere. My children Catherine, Anthony, and Jimmy were also considerate when it came to the special demands required of an AFIT student. Not only did they provide the silence that I required while studying at home (most of the time), but they also understood when dad could not make a PTA meeting or attend a little league game. Finally, I would like to thank my parents, Wayne and Mildred Belford. Without the values they instilled in me as a child, I would not have even made it to AFIT let alone complete the program.

Table of Contents

List of Figures

Abstract

Simulations which model the behavior of "real world" entities are often large and complex, and require frequent changes to the configuration. This research effort examines the benefits of using object-oriented techniques to develop a distributed simulation environment which supports modularity, modifiability, and portability.

The components of a Parallel Discrete Event Simulation (PDES) environment are identified and modeled using the Rumbaugh modeling technique. From the model, a prototype implementation of a Parallel Ada Simulation Environment (PASE) is accomplished using Classic Ada. A system interface for the Intel ipsc/2 Hypercube was developed to illustrate the concepts of modularity and portability. In addition, the prototype environment uses a filter which implements the basic Chandy-Misra time synchronization protocol.

Finally, To test the correct operation of the environment, a simple battlefield application model is developed. PASE is tested in the sequential mode on both a Sun Sparc station and the Hypercube. The ability to distribute across *n*-nodes is demonstrated using various configurations on the Hypercube. The parallel test demonstrates the ability for objects on separate processors to interact with each other by passing messages, and to execute events generated by remote objects in the proper time stamp order.

OBJECT-ORIENTED DESIGN AND IMPLEMENTATION OF A PARALLEL ADA SIMULATION SYSTEM

I. Introduction

1.1 Background.

The complexity of systems in today's Air Force coupled with the reality of shrinking budgets requires that a new way of doing business be found. State of the art modeling and simulation (M&S) provides a feasible solution. A report released by the United States Department of Defense (DoD) in March of 1990 identified simulation and modeling technology (SMT) as one of twenty technologies vital to ensuring the long-term qualitative superiority of our military forces (17 : 1). This technology has advanced to the point where modelers can now create incredibly realistic, extremely detailed models which can augment test and evaluation, support the acquisition process, facilitate intelligence gathering and support detailed engineering. One system under development to support M&S throughout the DOD is the Joint Modeling and Simulation System (J-MASS) (4 : 1). Unfortunately, as the complexity of the systems we wish to model increases, the time that it takes to run the simulation in a sequential environment can become prohibitive. For instance, the need for quick decisions on the battlefield and shorter acquisition times demand parallelization of the simulation process. This was apparent during Operation "Desert Storm" when the results of battlefield simulations could not be provided expeditiously enough to benefit strategic planning. In the future, the need for speed can be met if models developed using the J-MASS and other standards can be efficiently executed on parallel architectures such as the Intel IPSC/2 Hypercube. Parallel architectures offer the potential for a speedup of n-times with an n-processor computer.

1.2 Problem Definition.

a. The current parallel discrete event simulation (PDES) environment used at AFIT is SPECTRUM which was developed at the University of Virginia (8). SPECTRUM has been used at AFIT for student research with mixed success. Most students found SPECTRUM to possess a large learning curve, and perceived SPECTRUM to be limited in its ability to provide the needed simulation environment support (such as synchronization protocols) for their projects. Several simulations would not run under the original environment, and this required students to adapt SPECTRUM to meet their needs. This lack of a standard executive resulted in the inability to accurately compare and contrast results in different areas of research at AFIT. An object oriented discrete event simulation environment (TCHSIM) has been developed at AFIT that operates on top of the SPECTRUM testbed. TCHSIM has provided the ability to support two different application simulations, a battlefield simulation and a queuing system simulation, with no modification to the underlying layers. While this is a step in the right direction, TCHSIM is written in C and does not fully satisfy existing requirements. Current and future requirements for an Ada based object-oriented parallel simulation executive (environment) exist to support R&D and experimentation in the areas of parallel and distributed simulation at AFIT. Also, DOD and Air Force requirements exist for a way to parallelize existing J-MASS and other models. Failure of current systems to address these requirements mandate the exploration of various approaches to parallel simulation and the development of a new simulation executive.

b. The design and implementation of the Parallel Ada Simulation System (PASS) was influenced by the following issues/requirements imposed to resolve the deficiencies stated previously.

1) It had to be adaptable to the two new DOD standards for simulation and modeling systems: the Distributed Interactive Simulation (DIS) and the Joint Modeling and Simulation System (J-MASS).

2) The low-level machine interface had to be transparent to the modeler.

3) Parallel simulation issues with regard to the implementation of models on parallel architectures such as the iPSC/2 Hypercube and networked Sun SparcStations had to be explored.

4) The new executive had to be robust enough to support both the conservative and optimistic paradigms as well as sequential, parallel, or distributed operation.

5) The executive was implemented using an Object-Oriented Programming (OOP) language. Since Ada is an object-based language used for J-MASS simulation models and is mandated by the DOD, it was the language of choice. Ada 9X provides the properties of an OOP not currently supported by Ada, and it will be the implementation language when it becomes available. Classic Ada provides the extensions needed to make Ada object-oriented and is used in the interim. The similarities between the two should provide for a relatively simple conversion to Ada 9X when it becomes available.

1.3 Overall Thesis Objective.

The overall objective of this thesis was to research existing studies and previously unexplored areas dealing with parallel simulation, and by using OOA and OOD techniques develop the design for an executive to support concurrent execution of models similar to existing J-MASS models. Using Object Oriented Programming (OOP) the design was to be implemented as a prototype on the Intel iPSC/2 Hypercube. In addition, the executive should provide a testbed to support R&D and experimentation in the fields of parallel and distributed simulation by the faculty and students at AFIT in the future.

1.4 Approach.

1) The first step in my research effort was to gain a thorough understanding of the current parallel simulation environments being used at AFIT: SPECTRUM and TCHSIM. This knowledge was then applied to the development of the new executive. A literature review was also performed in the area of heterogeneous distributed simulations to gain insight into current technology and its applicability to AFIT's requirements.

2) A thorough review of the J-MASS and DIS requirements documents was performed to identify all requirements the new executive would have to comply with to allow adaptability.

3) An Object-Oriented Analysis was performed to model the executive by examining requirements, analyzing their implications, and restating them in an understandable fashion. This was done at a level of abstraction that stated *what* must be done without restricting *how* it is to be done avoiding implementation decisions. The OOA technique used to accomplish this was Rumbaugh's object modeling technique (16). Rumbaugh's technique was chosen because it is the method currently taught at AFIT.

4) An Object-Oriented Design phase followed OOA. At this stage, system design issues were resolved and the models from the previous stage were coded using Classic Ada, an Ada like pseudo code.

5) The final step was to actually implement a functional prototype of the executive on the Intel iPSC/2 Hypercube.

1.5 Sequence of Presentation.

This first chapter provides a brief introduction to the problem and the approach taken to solve it. Chapter 2 consists of a Literature Review which includes a general description of the circumstances leading up to the problem, and an analysis of the research that was conducted to solve the problem. Chapter 3 contains an enumeration and analysis of the Parallel Ada Support System (PASS) requirements, and the models developed using the Rumbaugh technique; the Object Model, the Dynamic Model, and the Functional Model. Chapter 4 presents the design decisions and the analysis which was performed to reach them. In addition, any issues regarding implementation of the design are discussed. The test results are provided in Chapter 5. The chapter contains a description of the test scenario and an analysis of the data collected from initial testing of the prototype executive. Finally, Chapter 6 provides conclusions and research recommendations. This chapter summarizes the results of this research effort and identifies areas where further study would be beneficial.

II. Literature Review

2.1 Introduction.

A literature review was performed to gain the background knowledge required to develop a parallel simulation environment. The areas related to this research effort are; parallel simulation, methods of time synchronization, simulation environments, the Joint Modeling and Simulation System (JMASS), the Distributed Interactive Simulation (DIS), Rumbaugh's *object-oriented modeling techniques*, and object-oriented programming.

2.2 Parallel Simulation.

Computer simulations help predict the weather, make tactical decisions on the battlefield, present evidence in a court room, investigate aircraft accidents, and artificially duplicate the actions of many other real life operations. Unfortunately, the size and complexity of many simulations demand execution in a parallel environment to avoid outdated information which results from the excessive cost in terms of the time associated with execution on a sequential architecture. For example, a General does not need to know the simulated results of an airstrike after the mission has been flown! Parallel Discrete Event Simulation (PDES), sometimes referred to as distributed simulation, deals with the parallel execution of discrete event simulations on parallel architectures (7 : 1). Partitioning a simulation into several logical processes and distributing them over multiple processors can provide a speed-up which approaches the number of processors used. A *discrete event simulation model* assumes that the simulated system only changes state at discrete points in simulated time (7 : 1). Interest in PDES continues to grow with the size and complexity of simulations in the fields of engineering, military operations, and others where execution on a sequential machine requires enormous amounts of time.

2.3 Time Synchronization.

Partitioning a real-world system into a set communicating sequential processes results in a network of physical processes (PPs) which communicate information by passing messages. Interactions between two entities *i* and *j* in the system are modeled as messages between processes *i* and *j* in the network. The *realizability condition* asserts that "the behavior of a PP at time *t* cannot be influenced by messages transmitted to it after *t*" (5 : 198). In a simulation model, a logical process (LP) accurately *simulates* the behavior of a real-world PP up to a given time *t* if the initial state, and all messages corresponding to the PP up to time *t*, are known. The communication granularity is "the amount of computation between consecutive message events within a process communicating with other parallel processes" (6 : 10).

A goal of parallel processing is to realize near linear speed-up. The designer should partition the simulation in a way which maximizes communication granularity. Since messages received from other LPs can influence the behavior of an LP, it is imperative that all messages be processed in increasing order of their respective timestamp to prevent causality errors. The following two sections examine the two basic time synchronization methods developed to insure the processing of messages in the correct timestamp order - the *conservative* approach and the *optimistic* approach.

2.3.1 Conservative Simulation Approach.

In the conservative approach, an LP may not advance its local simulation time (T_i) until it is certain that no message with a time stamp $t < T_i$ will arrive. One example of a conservative simulation approach is the basic Chandy-Misra synchronization protocol (5 : 199). There is an LP which corresponds to every PP. An LP_i can simulate a message from PP_i to PP_j by sending a tuple to LP_j.

The format for a sequence of tuples would be

$$(t_1, m_1), (t_2, m_2), \ldots, (t_N, m_N)$$

and it is required that

1) $0 \leq t_1 \leq t_2 \ldots \leq t_N$, *(monotinicity)* and

2) A PP_i must have sent m_N to PP_j at t_N, $N = 1, 2, 3, \ldots$ and

3) PP_i must have sent no other message to PP_j besides $m_1, m_2, \ldots, m_N$.

Simply put, each message sent by an LP must have a time stamp higher than the previous message with the first occurring after time 0. The flow of messages between any two LPs must be identical to the flow between the corresponding PPs up to time *N*. The local simulation time (T_i) of an LP_i at any point in the simulation is defined as the maximum time satisfying the following:

1) The LP_i must guarantee that all subsequent input messages received along each input line have t-components greater than or equal to T_i.

2) The LP_i cannot send any message on any of its output lines if the t-component is less than T_i. (If an LP_i can guarantee that the t-components of all subsequent output messages will be greater than T_i, it can avoid sending a message out on every output line each time).

During the simulation, LPs alternate between computing and waiting to communicate. Whenever the LP is waiting to communicate it must follow these rules:

1) The LP waits to receive messages on all input lines whose clock values equal the LP clock value.

2) The LP must wait on all output lines on which there is a message to be sent.

After receipt or transmission of a message, the LP enters its computational phase where it determines which set of lines to wait on next according to the waiting rules. The LP updates its local clock and determines any outgoing messages that it must send up to the new time if all incoming messages have a time stamp greater than the LP time. Once the LP accomplishes this, it waits according to the rules given above.

The occurrence of deadlock is a major concern with the basic Chandy-Misra method. The following conditions are sufficient to indicate that a set of LPs are deadlocked (13 : 51) :

1) Every LP in the set is either waiting to receive a message or has terminated;

2) At least one LP in the set is waiting to receive;

3) For any LP in the set that is waiting for a message, there is no message in transit from any other LP in the set. Circular wait exists if the above conditions are met, and none of the LPs can carry out any further computation.

One solution to the problem of deadlock is the use of null messages. This approach entails sending a null message in the absence of messages. The null message (*t, null*) indicates to the receiver that the sender will not send any messages in the future with a t-component less than *t*. An LP treats the reception of a null message in the same manner as any other message: the receiving LP updates its internal state, including the local clock value, and sends any messages if required. Although the use of null messages prevents the simulation from deadlocking, not all null messages sent are necessary and the number of null messages sent can have a significant effect on the simulation time. There are variations on the null message approach, such as delaying transmission of the null message and allowing deadlock and then recovering, which Misra has explored (13 : 60).

2.3.2 Optimistic Simulation Approach.

Another approach used for time synchronization is the optimistic method where an LP uses techniques to detect and recover from causality errors rather than strictly avoiding them (7 : 17). In contrast to the conservative methods, the optimistic approach does not wait until it is safe to proceed; instead the process runs until the optimistic strategy detects an error, at which time the strategy invokes a procedure to recover. An advantage of this approach is that it allows exploitation of parallelism in situations where causality errors can occur but do not.

One example of an optimistic protocol is Time Warp. Time Warp follows the Virtual Time paradigm, where virtual time is synonymous with the local simulation time. Time Warp identifies a causality error when an LP receives a message with a time stamp less than the local simulation clock. When this situation occurs, the LP rolls back the effects of events that it processed with a time stamp greater than the new event. There are two effects which require rollback; modification of the LP's state, and the premature transmission of messages to other processes. By saving the LP's state periodically, it is possible to rollback to a safe state by restoring the old state vector. An LP uses message cancellation by sending negative or anti-messages to any processes that it sent premature messages to. If the receiving LP has not processed the premature message it destroys it. Otherwise, the process rolls back to its state prior to executing the premature message. The LPs repeat the message cancellation procedure recursively until they have corrected all of the effects of the erroneous message. There are two approaches used for canceling messages; the aggressive approach and the lazy approach. The aggressive approach as described above sends anti-messages immediately after an LP detects a causality error. The lazy approach on the other hand waits to see if reexecution of the computation regenerates the same messages. If the LP generates the same messages, there is no need to cancel the previously sent message. If the local simulation clock exceeds the time stamp of the anti-message before the LP regenerates the previously sent message, the LP processes the anti-message.

2.4 Simulation Environments.

Webster's dictionary defines an environment as the circumstances, objects, or conditions by which one is surrounded. A simulation environment can be thought of as the circumstances, objects, and conditions which make up the interface between the simulation programmer and the computer hardware. The simulation environment provides a level of abstraction which frees the programmer from dealing with the machine level details and allows efforts to be focused on the simulation itself. A common simulation

environment also provides the means to accurately and efficiently evaluate a variety of algorithms and protocols on a given application (15 : 325). Currently, there are two parallel simulation environments in use at AFIT: The Simulation Protocol Evaluation on a Concurrent Testbed with ReUsable Modules (SPECTRUM) and TCHSIM. Both environments are written in the C programming language and are hosted on the Intel Hypercube. An overview of both environments is provided below.

2.4.1 SPECTRUM.

SPECTRUM is a multi-layered system designed to provide a testbed on which various parallel discrete event simulations can be designed and evaluated in a common environment. SPECTRUM supports a model similar to that defined by Jayadev Misra (13 : 40-41) consisting of a set of communicating Logical Processes (LP). An LP is meant to include all of the code required to form an independent process to include the application, lp_manager, and node_manager. A physical processor, or node, can contain one or more LPs. The layers, as shown in Figure 1, are the application layer, lp_manager, and the machine layer.

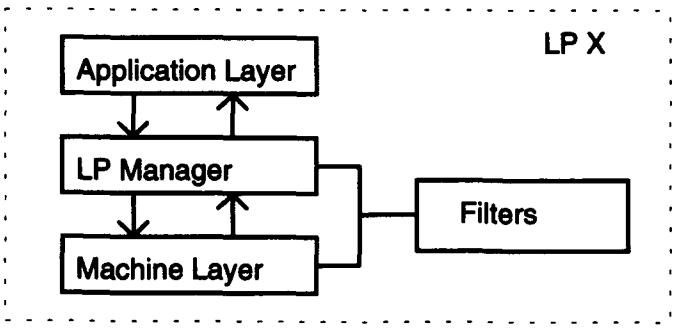


Figure 1. Structure of the SPECTRUM testbed.

The application layer, which consists of the user supplied program, is machine independent and makes calls to the lp_manager. The lp_manager provides the interface between the application program and the machine layer and includes support for initialization, clock advancement, and message management. The lp_manager maintains an input queue of messages from other LPs in simulation time-stamp order, which allows LPs to communicate. The machine layer is application independent and consists of a node manager. The node manager acts as the interface between the lp_manager and the hypercube and provides functions for memory management and communication. The filters exist to allow implementation of various simulation time synchronization protocols.

2.4.2 TCHSIM.

TCHSIM was developed to provide a general discrete event simulation environment which allows experimentation with several different models without having to re-implement the basic structure each time (10 : 1). The approach taken in the development of TCHSIM was to start with a general sequential environment and to incrementally add capabilities which provide for parallelism. The parallel version of TCHSIM is depicted in Figure 2.

At the top of the simulation environment is the simdrive.c file which acts as a start-up and initialization module (driver). The file <application>.c is the user written application program which is the implementation of the simulation model. A file "simshell.c" has been developed to give the programmer a starting point in the development of application software for TCHSIM/SPECTRUM. The file interface1.c contains calls to several object-oriented functions; event, clock, next event queue, and general support. In addition, it provides an interface to the SPECTRUM logical process and node managers.

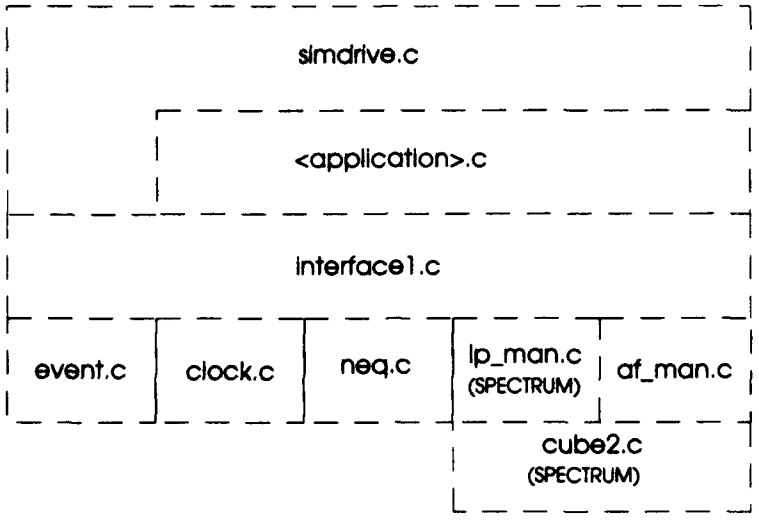


Figure 2. Structure of TCHSIM Version 3.

2.5 Joint Modeling and Simulation System

The Joint Modeling and Simulation System (J-MASS) is a modeling system designed to support engineers, model developers, analysts, and decision makers (4 : 1). Modeling and Simulation (M&S) has become an important aspect of the analysis and decision process in the DOD, and J-MASS is a tri-service program intended to provide a standard environment to carry out activities related to M&S. Currently the first models developed in compliance with the J-MASS standard architecture are complete. To date these models have only been executed sequentially on a single processor. The J-MASS software system consists of two basic parts: the Simulation Support Environment (SSE) and the Software Structural Model (SSM).

The SSE allows modelers to access libraries of existing components which are linked together to create models. J-MASS models are based on "real world" systems (objects), such as a tank or an airplane, which may be broken down into components such as engine, fire control, and radar subsystems or engine, flight-controls, and munitions subsystems respectively.

The Software Structural Model provides a standard design methodology which is applied to all modeling components which are developed for and placed in the J-MASS modeling library. The SSM design methodology was derived from the Object-Connection-Update (OCU) model and follows the application of object based requirements analysis to a given problem space which produces a hierarchical partitioning of the object to be simulated (12 : 7). The object-based requirements analysis produces an object-based graphical representation of the system to be modeled using Object-Oriented Design with Assemblies (OODA) diagrams. Objects from the J-MASS OODA drawings are mapped onto a standard SSM for each object class (12 : 16). The software is stored as reusable templates which contain a hard coded portion which does not change from model to model, and a soft coded portion which changes depending on the object being modeled. The underlying premise of the SSM methodology is to identify, graphically document, and textually document recurring software structural patterns. The requirements for J-MASS are further enumerated in Chapter 3.

2.6 Distributed Interactive Simulation.

DIS is a fairly new system that provides the integration of computers and communications by defining fully interoperable standards and protocols (19 : 1). The system is distributed in the sense that it allows for geographically separated simulations (cells) which are hosted on remote computers connected by a communication network to create a shared synthetic environment, i.e., no central computer. It is interactive because entities from the geographically separated simulations are electronically linked and may act together and upon each other by passing messages in the prescribed format of Protocol Data Units (PDUs) (see Figure 3).

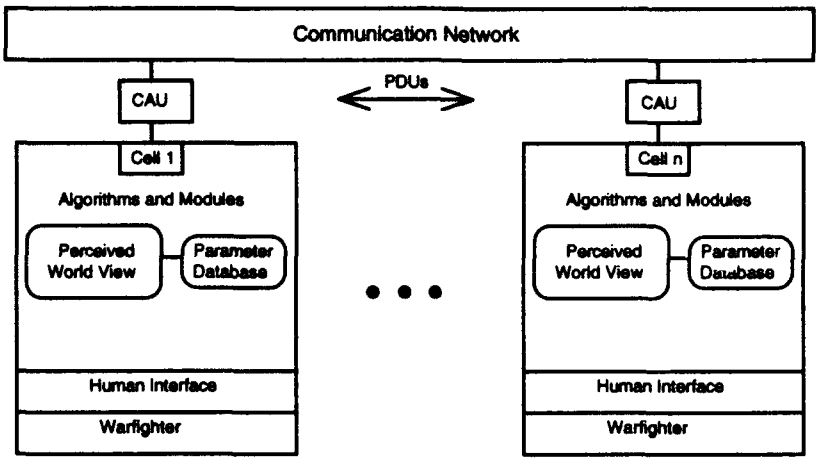


Figure 3. DIS Reference Model.

Each cell contains one or more homogeneous simulation entities. The Cell Adapter Unit allows simulations not compliant with the DIS standard, such as PASS, to participate in an exercise by translating the messages between the two protocols. Each cell maintains a perceived world view of local entities as well as entities of interest from other cells which are within its sight by use of dead reckoning algorithms. The actual position of entities from other cells is updated only when a predefined threshold is exceeded with respect to the difference between an entity's perceived position and its actual position. This along with data compression minimizes network traffic. Time and space accuracy requirements depend on the type of entity. For example, air-to-air combat involves high performance vehicles and weapon systems which require very high PDU rates to achieve the desired positional accuracy. There are basically three classifications of simulations as follows:

- Live - Operations with real equipment in the field.
- Virtual - Systems and troops in simulations fighting on synthetic battlefields.
- Constructive - War games, Models, and Analytical tools.

DIS is examined more closely in Chapter 3.

2.7 Object Oriented Analysis and Design.

The concept of Object Oriented development allows a software engineer to model a system in conceptual terms which closely match the physical system being modeled. There are several accepted methods of accomplishing this in use throughout the industry. The method taught at AFIT and chosen for the Parallel Ada Simulation System is the Rumbaugh method (16 : 1).

An object-oriented analysis and object-oriented design facilitates the implementation of a system using an object-oriented programming language. Object-oriented analysis entails modeling the system to provide an abstraction for the purpose of better understanding it before an attempt is made to build it. The Object Modeling Technique (OMT) combines three views of modeling the system (16 : 17). The *object model* represents the static, structural, "data" aspects of the system. The *dynamic model* represents the temporal, behavioral, "control" aspects of the system. The *functional model* represents the transformational, "function" aspects of a system. The object-oriented design entails decisions about organizing the system into subsystems, allocation of subsystems to software components, management of data stores, access to global resources, and control of the system software.

The object diagram is a graph whose nodes may be viewed as object classes and the arcs between the nodes are the relationships between the classes. An object class is described by its attributes and methods as shown in Figure 4.

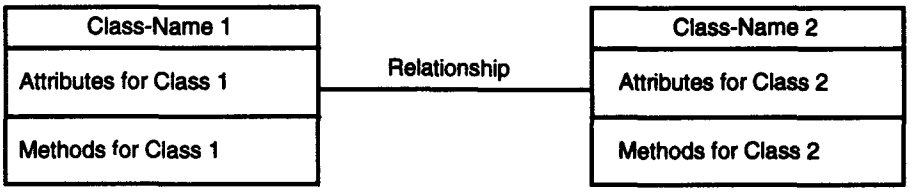


Figure 4. Rumbaugh's method of object modeling notation for classes.

The dynamic model contains state transition diagrams which describe the aspects of a system that change over time. The state diagram is a graph whose nodes are states and arcs are transitions between those states caused by events. The notation for a Rumbaugh state transition diagram is shown in Figure 5.

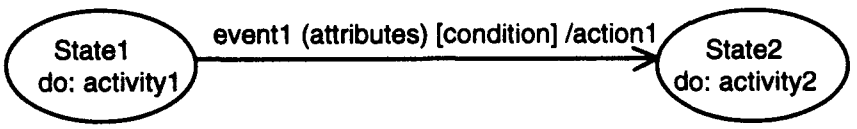


Figure 5. Rumbaugh's method of notation for unstructured state diagrams.

The functional model describes the data value transformations within a system. The functional model is made up of data flow diagrams. A data flow diagram can be thought of as a graph whose nodes are processes and whose arcs are data flows. The notation for a Rumbaugh data flow diagram is shown in Figure 6.

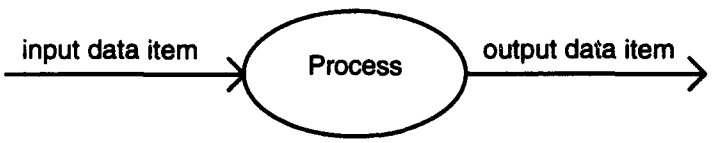


Figure 6. Rumbaugh's method of notation for data flow diagrams.

2.8 Object Oriented Programming Languages.

Object Oriented Programming (OOP) techniques have proven to be a powerful tool which leads to well-structured, flexible, and reusable software. Classic Ada is the first

language to provide true object oriented programming to the Ada developer (18 : 1). While Classic Ada is an extension to the Ada language, Ada 9x will provide the programmer with object oriented features contained in the language itself.

2.8.1 Classic Ada.

Certain principles are fundamental to Object Oriented Programming. Classic Ada supports the following object-oriented principles (18 : 2):

- *Information hiding* which protects the implementation of an object from exterior manipulation, increasing modularity and reliability, and decreasing coupling.
- *Data abstraction* which encapsulates the data representation and its operations into an object.
- *Dynamic binding* which determines the operation to be invoked when a message is actually sent to an object.
- *Inheritance* which enables users to easily create objects that are similar to existing objects, yet can be tailored to their needs.
- *Polymorphism* which enables different types of objects to respond to the same message in different ways.

The basic element in Classic Ada is the object. The object can be described as either a class object or an instance of a class object. The class object describes a set of instance variables and instance methods which are common to all instance objects of that class. For example, a vehicle class would describe the attributes and actions that are common to all vehicles as shown in Figure 7. Both the Aircraft and Tank instances of the class Vehicle inherit its attributes and methods. Through specialization, the object instances can identify the differences between themselves and other vehicles by adding variables and methods, or redefining the inherited class methods.

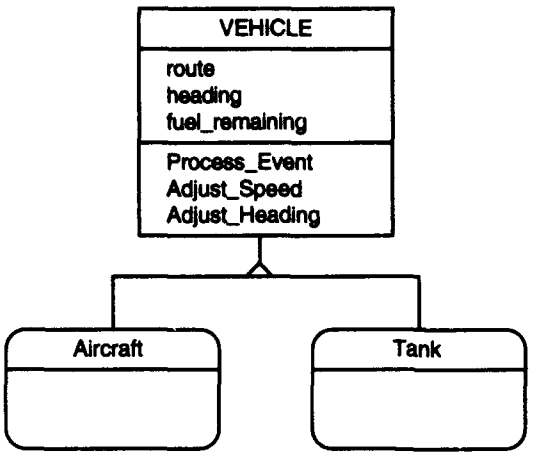


Figure 7. Class and subclass Objects.

Interaction between objects is accomplished by sending messages. The object's internal state and structure is not visible to the outside world. The only mechanism for requesting that an object perform some action or provide some information is sending a message to the object. The basic format of the message is

SEND (Object_ID, Method, Parameter List).

Object_ID identifies the object that the message is intended for, *method* identifies the instance method to be invoked, and the *parameter list* identifies the parameters to be passed in and returned.

In summary, Classic Ada contains class objects, instance objects, instance variables, and instance methods. Instance objects inherit the attributes and methods of their super class. Communication between objects takes place using messages which request that an operation be performed upon an object.

2.8.2 Ada 9x.

Ada 9x is a revision to the Ada 83 programming standard which increases the flexibility of Ada while retaining Ada's inherent reliability (1 : i).

Enhancements to Ada 9x include

- *Object Oriented Programming* which provides full OOP facilities giving the flexibility of programming by extension. Extension is the ability to add to existing classes of objects.

- *Hierarchical Libraries* The library mechanism now takes a hierarchical form which is valuable for the control and decomposition of large programs.

- *Protected Objects* The tasking features of Ada are enhanced to incorporate an efficient mechanism for multi task synchronized access to shared data.

2.9 Summary.

Previous research in the field of Parallel Discrete Event Simulation shows that parallel processing can effectively speed up the execution of simulations. The design of a simulation environment which complies with the objectives of this research required knowledge of existing PDES environments to identify common components (objects) such as a clock, next event queue, and protocol filter. The existing PDES environments examined were SPECTRUM and TCHSIM. In addition to identifying components of a general PDES, the lessons learned from the two existing environments at AFIT provided the insight necessary to avoid problems sometimes associated with current simulations such as the lack of a standard testbed and difficulty in modifying configurations. Examination of the J-MASS and DIS standards provided the knowledge required to understand what is required to interface with each. The detailed interface requirements are in Chapter 3. Two methods of time synchronization, the conservative approach and the optimistic approach, were researched and the results used to choose and develop a protocol for this research effort. Finally, object-oriented techniques for system analysis, design, and programming were studied to provide the tools necessary to develop and implement a prototype simulation environment using the Ada programming language.

III. Requirements Analysis

3.1 System Overview.

The purpose of the Parallel Ada Simulation System (PASS) is to provide a platform on which models similar to J-MASS models can be executed in parallel. The PASS allows modelers to develop models without concern for the low level machine interface, and it provides the flexibility to allow experimentation with various configurations as described in Section 3.2. The PASS consists of two major subcomponents. The Parallel Ada Simulation Model (PASM) provides code templates that a modeler can use to develop models for the Parallel Ada Simulation System. The Parallel Ada Simulation Environment (PASE) provides the PDES services to support execution of the models.

This chapter outlines the specific engineering requirements and the corresponding object-oriented analysis for a simulation system that will fulfill the purpose stated above. In addition, the requirements which must be met to interface with the Distributed Interactive Simulation (DIS) are given consideration. It should be noted that although the DIS requirements are delineated and considered during the analysis and design, the implementation is left for future enhancements to the system.

3.2 Engineering Requirements.

3.2.1 Simulation Environment Requirements.

The simulation environment requirements are the general requirements which the PASE must satisfy to support the activities associated with the execution of a generic Parallel Discrete Event Simulation (PDES). These requirements are derived from the experience gained by previous efforts at developing simulation test beds at AFIT; namely SPECTRUM, TCHSIM, and Project-Q (9).

3.2.1.1 Simulation Configuration.

To allow experimentation with parallel issues such as using various synchronization protocols and data structures, the simulation environment shall be designed in a modular fashion to allow changes to the configuration to be made easily.

3.2.1.2 Simulation Control.

Simulation control is required to provide the following services:

1) Initialize internal control of the simulation based on user inputs (runtime, protocol, etc).

2) Provide a means for starting and stopping the simulation.

3) Control communication between entities.

- determine if an entity is local or remote
- determine optimum event execution location
- locate remote objects
- schedule remote events
- transmit an event to a remote LP
- receive an event from a remote LP
- trade object state data between LPs

4) Store simulation results.

5) Distribute simulation across *n* nodes.

3.2.1.3 Clock.

A local clock is required to keep track of the LP's notion of simulation time. Methods should be provided to initialize, set, update, and read the value of the clock.

3.2.1.4 Next Event Queue.

A local next event queue (NEQ) is required to maintain events in the proper time stamp order. The operations to be performed on the NEQ are as follows:

1) Initialize_NEQ - initializes NEQ.

2) Add_Event - Adds a new event to the NEQ.

3) Get_Event - Retrieves the event at the top of the NEQ.

4) Display_Queue - Displays the events currently on the NEQ.

5) Count_Events - Returns the number of events on the NEQ.

6) Minimum_Time - Returns the time of the event at the top of the NEQ.

7) Maximum_Time - Returns the time of the event at the bottom of the NEQ.

8) Equal_Time - Returns the number of events with simultaneous earliest simulation times.

9) Max_Size - Returns the maximum NEQ size during execution of the simulation.

3.2.1.5 Migrating Objects Between LPs.

The ability to move an object dynamically from one LP to another should be provided to accommodate lightly loaded processors and to minimize communication delays. This can be accomplished by including algorithms in the environment to determine the optimal distribution of logical processes across physical processors based on communication dependencies and processor utilization. The dynamic migration of objects between LPs is left as a future enhancement.

3.2.1.6 Time Synchronization.

To insure that the simulation executes in the proper time sequence, a method for event synchronization shall be implemented. The method used should be transparent to the application. The method chosen for the prototype version of the Parallel Ada Simulation Environment will be the basic Chandy and Misra conservative approach.

3.2.1.6.1 Conservative Approach.

The exact requirements for this technique are described in Chapter 2. Basically, an LP may not advance its local simulation time (T_l) until it is certain that no message with a time stamp $t < T_l$ will arrive. The safe time is defined as the smallest time stamp of the last received messages across all of the LP's input channels. To prevent deadlock, some type of null message scheme is also required.

3.2.1.6.2 Aggressive Approach.

The aggressive (optimistic) approach does not wait until it is safe to proceed; instead the process runs until an optimistic strategy detects an error, at which time the strategy invokes a procedure to recover. To accomplish this, a mechanism is required to save the state of the system, and a strategy is used to "rollback" to a previous system state. This technique is left as a suggested future enhancement to the system.

3.2.2 External Interface Requirements.

The characteristics of both the Joint Modeling and Simulation System (J-MASS) and the Distributed Interactive Simulation (DIS) are examined to allow the Parallel Ada Simulation System (PASS) to be structured in a manner that will allow future compatibility with both.

3.2.2.1 Joint Modeling and Simulation System.

This section contains the general requirements which must be met to execute J-MASS models or those which are compliant with the J-MASS standard. J-MASS is made up of three major components; the Simulation Runtime Agent (SRA) executive, the executable processes (teams), and the Simulation Runtime Interconnect Backplane (IBP) (12 : 9).

The following paragraphs present the responsibilities of each major component and its subcomponents as given in the J-MASS Software Development Plan for the Software Structural Model (12 : 17-20).

3.2.2.1.1 SRA Executive.

The following components belong to the executive level of the SRA:

- Process Controller
  - — Accepts simulation information from the SSE
    - ⇒ scenario information
    - ⇒ distribution information
  - — Uses the distribution information to
    - ⇒ extract definition of teams
    - ⇒ spawn team processes
    - ⇒ record process IDs for control and monitoring during execution
  - — Initiate the Scenario Manager, Jounalizer(s), and Synchronizer(s)
  - — Start and stop simulation based on user commands
- Scenario Manager
  - — Uses the scenario information to create and initialize team players
    - ⇒ player id
    - ⇒ player class
    - ⇒ initial spatial state
    - ⇒ configuration data id
    - ⇒ relationship to other players
- Spatial Manager
  - — Maintains spatial information for each player throughout simulation execution
    - ⇒ position
    - ⇒ orientation
    - ⇒ linear velocity and acceleration
    - ⇒ rotational velocity and acceleration
  - — Maintains the spatial relationship between players

- Synchronizer
  - — Maintains the global simulation clock
    - ⇒ determines when to advance simulation time and step size
  - — Sychronizes the execution of all teams
    - ⇒ execution in proper timestamp order

3.2.2.1.2 Simulation Runtime Interconnect Backplane.

The Simulation Runtime Interconnect Backplane (Communications Area) extends across all nodes in a simulation and represents the paradigm responsible for message traffic between the SRA executive components and Team components. The way in which messages are communicated is hardware dependent and the requirements for the host architecture are contained in Section 3.2.2.3.

3.2.2.1.3 Team (Process) Components.

The following components belong to the Team level of the SRA:

- Services Interconnect Backplane (Communications Area)
  - — Provides simulation services for model components
    - ⇒ Inter-player communication
    - ⇒ Intra-player communication
    - ⇒ communication between players and other team components
- Environment
  - — Retains state information of all other players
    - ⇒ calculates relative state vectors between players
    - ⇒ calculates relative state vectors between players and terrain features
  - — Generates signals initiated by RF and electro-optical/infrared players
  - — Applies any environmental affects on the signal between sender and receiver
  - — Communicates spatial information with Spatial Manager
  - — Exchanges signal information with the players

- Data Management Package (DMP)
  - — Responsible for all aspects of data access management within a simulation
  - — Acts as data interface between the SRA components and the model
  - — Stores information about modeling components and instances of these components
- Journalizer
  - — Collects data during simulation execution from the DMP
    - ⇒ to provide a means for simulation saved-point repositioning
    - ⇒ to provide a means for simulation recovery from unusual conditions
    - ⇒ to provide a means for recording and playback of all or part of the simulation
- Characteristic Manager(s)
  - — Responsible for type conversions when accessing standardized data from the DMP

3.2.2.2 Distributed Interactive Simulation.

This section contains the requirements which must be met for a simulation to participate in the shared synthetic environment known as the Distributed Interactive Simulation (DIS). The local simulation is viewed by DIS as one of many cells which may interact with each other over a virtual network. This interaction takes place by passing messages via Protocol Data Units (PDUs) between application processes.

3.2.2.2.1 Distributed Simulation Management.

The Distributed Simulation Manager (DSM) is responsible for entity/exercise management and data management. The DSM performs this function by communicating via simulation management PDUs. The DSM is responsible for the following four functions with respect to entity/exercise management:

- Creation and initialization of a new entity/exercise.
- Changing an existing entity's parameters.
- Starting or stopping an entity/exercise.
- Removing an entity from an exercise.

In addition, the DSM is responsible for the following functions with respect to data management:

- Request for data.
- Setting or changing internal state values.
- Entity reconstitution.

3.2.2.2.2 Dead Reckoning.

A method must be implemented for estimating the position/orientation of entities based on previously known position/orientation and predefined estimates of time and motion. Specifically, the host environment must

I. Maintain a high fidelity model of each local entity (actual position) and a low fidelity model of each local entity (dead reckoned position). Whenever the high fidelity model differs from the low fidelity model by more than the set threshold, the environment will update the entity's dead reckoning to the actual position as well as communicate an entity state PDU to other entities for them to update their dead reckonings.

II. Maintain a dead reckoned model of all other entities of interest that fall within a predefined value for sight or range. This dead reckoned position will be used to track each entity's position. Smoothing techniques should be employed if the capability exists to display an entity's position to prevent sudden changes from one location to the next. The dead reckoned models shall be updated with the most current information for an entity as it is received.

The dead reckoning algorithms contained in Appendix I of the DIS military standard (11 : 163) shall be used to approximate the position of entities of interest.

3.2.2.2.3 Cell Adapter Unit.

The cell adapter unit acts as an interface between two cells in an exercise by performing two main functions:

I. The front end is responsible for message filtering between cells.

- PDUs may be eliminated based on an entity's location on the battlefield.
- Provides a reduction in intercell traffic.
- Reduces entity I/O processing.

II. The back end acts as a message translator.

- Converts outgoing messages to non-standard DIS cells.
- Converts incoming messages from non-standard DIS cells.

3.2.2.3 Hardware Interface.

The hardware interface will be generic in the sense that it can be instantiated for various architectures (i.e. Intel Hypercube, Shared Memory, Network of Sun Sparc Stations). Based on the results of a comparison of the communication costs associated with various architectures performed for another graduate class, the prototype system will be hosted on the Intel IPSC/2 Hypercube. The following subsection describes the requirements of the hardware interface - the node manager.

3.2.2.3.1 Node Manager.

The node manager will provide a modular interface to the machine dependent layer. A node is meant to refer to a physical processor, whether it is one cpu from a cube architecture, one SUN workstation from a group, or a single CPU in a shared memory network. The node manager provides communication services between nodes in a manner that is transparent to the simulation program. The prototype host will be the AFIT hypercube.

The AFIT IPSC/2 Hypercube is a Multiple Instruction Multiple Data (MIMD), distributed memory (loosely coupled) architecture. Each node operates as an independent system; interaction between nodes requires messages to be passed. Since passing messages represents overhead, the simulation should be decomposed and distributed in a manner

which minimizes communication between processors. In addition, the simulation should be decomposed in a manner which distributes the workload evenly across the number of processors available. The method of accomplishing this for the prototype system will be by examining the players in the simulation and manually distributing them across the nodes of the Hypercube to minimize both communication overhead and processor imbalance. A recommended future enhancement will be to automate this process. The hypercube offers a flexible connection scheme which provides a potential for high bandwidth while reducing the maximum distance that messages must travel. The hypercube interconnection network is shown in Figure 8. The maximum distance the message must travel, for example from node 0 to node 7, is three "hops" as opposed to a distance of seven in a linear array network.

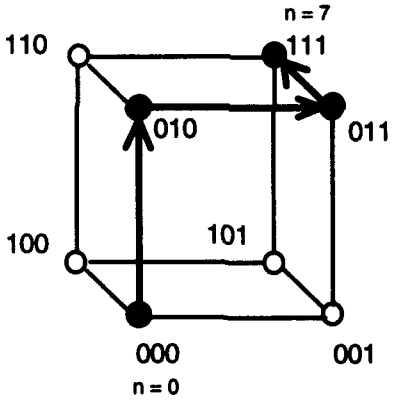


Figure 8. Message Passing Network based on the Hypercube Connection.

The node manager will send messages to and receive messages from other nodes on the cube when required and maintain an input queue of messages received from other nodes in simulation time-stamp order. Messages will be processed using the node_ipsc library calls

- CPROBE - block for an incoming message.
- CRECV - receive a message and wait for completion before proceeding.
- CSEND - send a message and block until execution is complete.

3.3 Object-Oriented Analysis (OOA).

This section contains the object oriented analysis of the engineering requirements provided in Section 3.2. The purpose of this analysis is to model the requirements of the system in a precise, concise, and understandable fashion. The analysis model consists of object, dynamic, and functional models and serves as the basis for the design and implementation.

3.3.1 Information (Object) Model.

Three separate object models are used to represent the Parallel Ada Simulation System. The Parallel Ada Simulation System (PASS) object model shows the associations between the three entities in a simulation: the application, the environment, and the host architecture. The other two models represent the components of PASS which were developed as part of the prototype system. The Parallel Ada Simulation Model (PASM) object model represents the structure of an application designed to run in the PASS and the Parallel Ada Simulation Environment (PASE) object model describes the environment that provides the services required for a PDES.

3.3.1.1 Parallel Ada Simulation System Object Model

The object model for the overall PASS is shown in Figure 9. Solid lines are used to represent the objects developed as part of this thesis effort. The PASS consists of two major components (objects); the application, and the environment in which the application runs (PASE). There are two types of applications, those which meet the PASE interface requirements and others which do not. The Parallel Ada Simulation Model (PASM) falls into the first category. Models which comply with other standards, such as DIS and J-MASS, but are not compliant with the PASE interface can still be executed in the PASE environment provided an application interface is written to map their interface requirements. PASE

executes on a given architecture and may be replicated on multiple nodes. The user is responsible for selecting the application and providing the application interface if one is required.

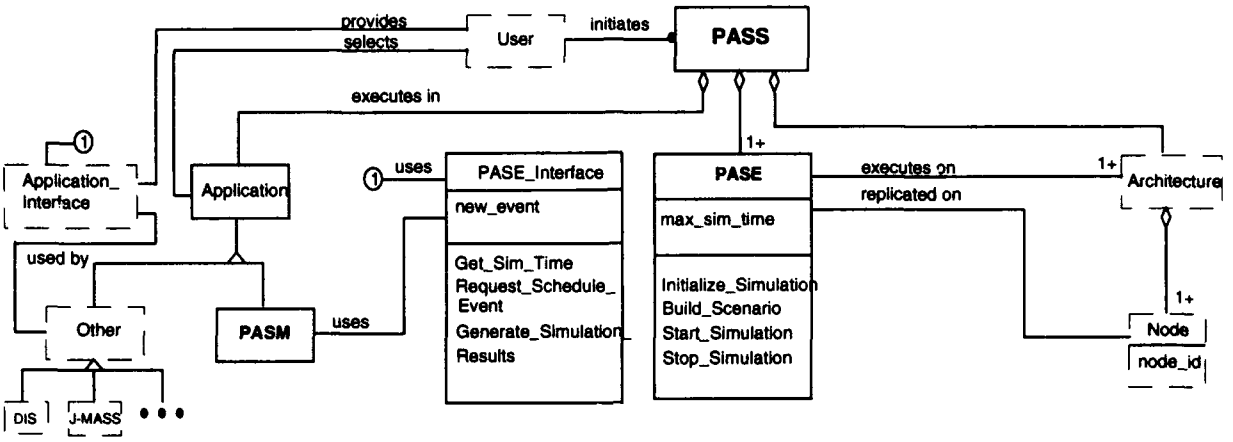


Figure 9. Object Model for the Parallel Ada Simulation System (PASS)

3.3.1.2 Parallel Ada Simulation Environment Object Model

The object model for the PASE is shown in Figure 10. The environment is an aggregate of four major components; the I/O Manager, Synchronizer, Logical Process, and Node Manager. The PASE object is responsible for initializing the environment, building the simulation scenario, starting execution of the simulation, and stopping the simulation when either the maximum simulation time has been reached or all events have been processed. Scenario information will be read from an application file and simulation data will be written to an output file to allow post simulation performance analysis. The Synchronizer object is an aggregate of the components required to provide the services necessary to ensure events are executed in the proper time stamp order. Subclasses, which infer the "is-of" relationship, exist for the I/O_Manager, the Protocol_Filter, and the Node_Manager objects. A Remote_Node object is used to maintain input and output channel times from other nodes in a simulation.

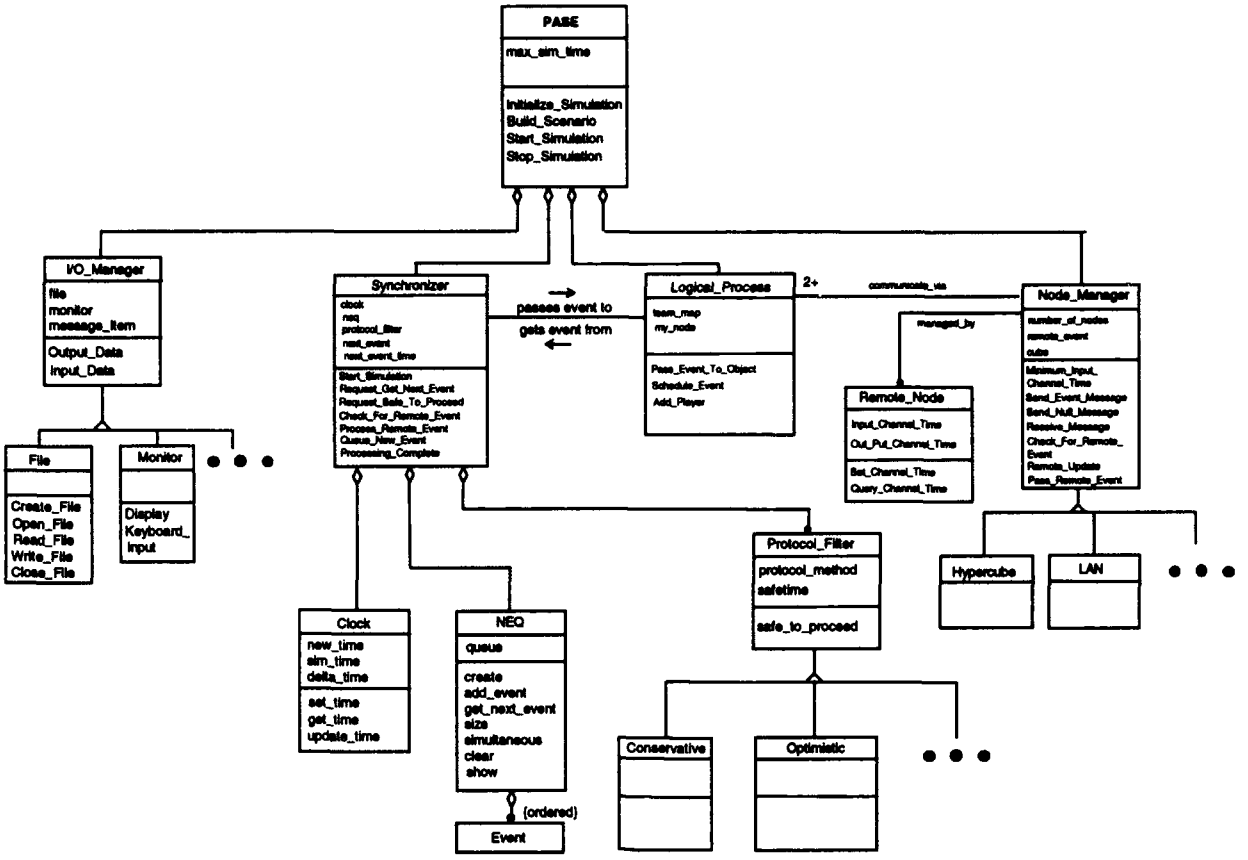


Figure 10. Object Model for the Parallel Ada Simulation Environment (PASE).

3.3.1.3 Parallel Ada Simulation Model Object Model

The object model for the PASM is shown in Figure 11. Each Parallel Ada Simulation Model is an aggregate of one or more teams consisting of many Player objects. Each Player object has associated with it an Event Class which describes the possible events that Player must be able to process. In addition, each Player object is associated with an Entity Location which describes the Player's location in three dimensional space. A Player can be of type Vehicle or Environment. Vehicles operate in a given Environment. A Vehicle also has a Route associated with it which is an ordered set of Route Points. Each Route Point is described by a location in three dimensional space. Although others may exist, two possible types of vehicles are an Aircraft and a Tank. Each type of vehicle may have associated component classes such as Engines, Missiles, Flight_Controls, etc... .

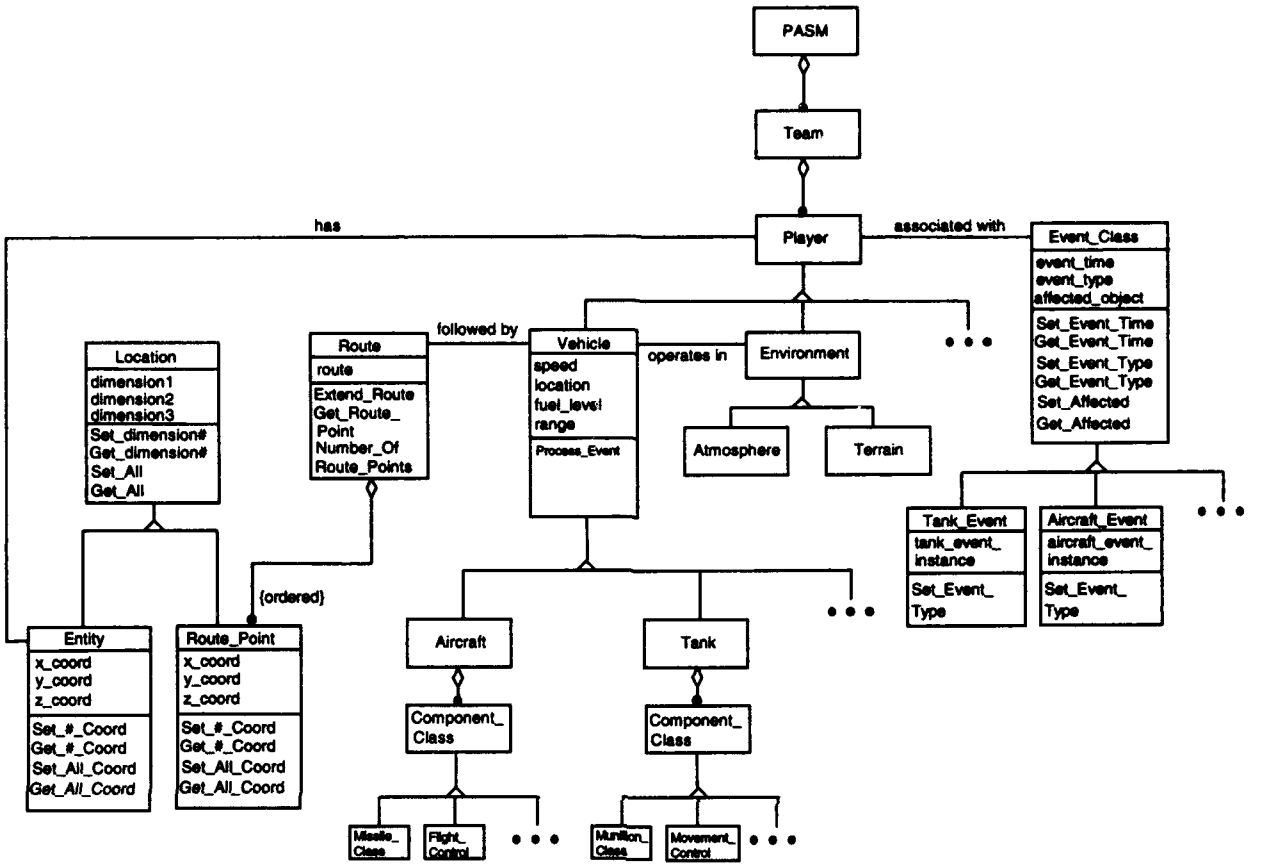


Figure 11. Object Model for the Parallel Ada Simulation Model (PASM).

3.3.2 Dynamic Model.

The dynamic model is concerned with the time and state change aspects of a system. The changes in state that an object incurs do to external stimuli (events) are represented by State Transition Diagram (STD). This section provides state transition diagrams for PASE and its major components.

3.3.2.1 STD for the PASE Object

The Parallel Ada Support Environment has four states associated with it. When the user executes the PASE_MAIN file, PASE enters the Initializing Simulation state. In this state

the PASE object creates all of the objects immediately subordinate to it and they in-turn create the objects immediately below them until all objects in the environment have been created. In addition, after each object is created, it must initialize all instance variables to their starting values. After the PASE object completes the initialization it enters the Building Scenario state. In this state, the PASE object accesses an application information file and creates the objects associated with the given scenario. As each object of the Parallel Ada Simulation Model is created, the object creates any associated objects and initializes its instance variables. This is repeated i ratively until the entire Model Scenario has been built. After the environment and model are created and initialized the simulation begins execution. The simulation remains in this state until the maximum simulation time is reached or all events have been processed. Under either of these two conditions the simulation enters the Stopping Simulation state where the objects are iteratively deleted and their instance variables finalized (memory deallocated). The State Transition Diagram for the PASE object is shown in Figure 12.

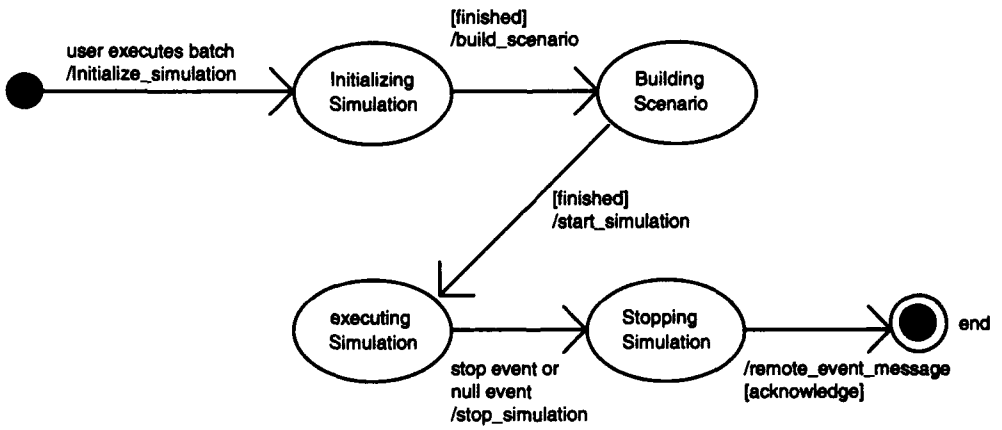


Figure 12. STD for the Parallel Ada Simulation Environment (PASE).

3.3.2.2 STD for the I/O Manager Object

The State Transition Diagram for the I/O Manager class is shown in Figure 13. During the initialization state, the I/O Manager creates any subordinate classes which handle the

I/O to particular devices such as a file or the standard devices. From there it transitions to the Idle state where it remains until an I/O message is received. The type of device then drives which subclass is used to process the I/O data.

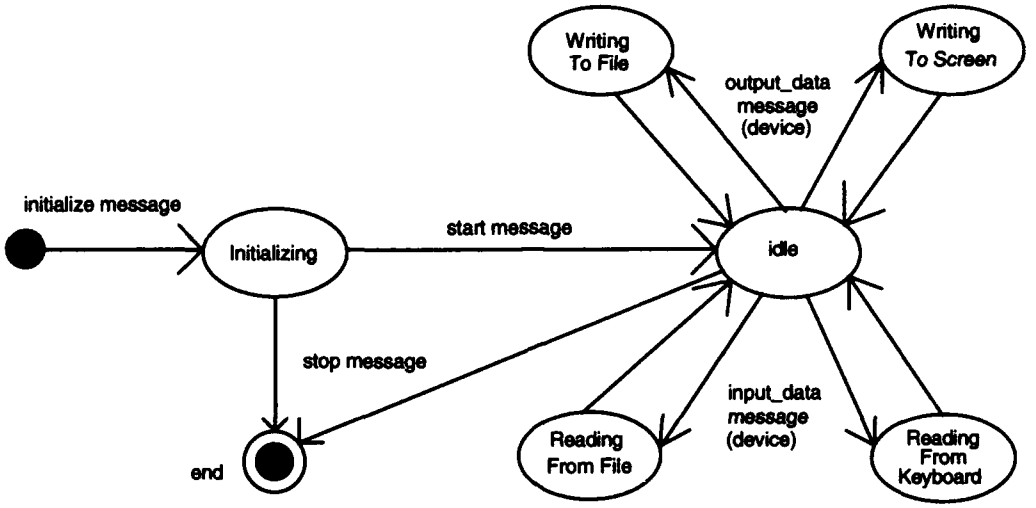


Figure 13. STD for the I/O Manager Class.

3.3.2.3 STD for the Synchronizer Object

The State Transition Diagram for the Synchronizer object is shown in Figure 14. When a message is received to initialize, the Synchronizer creates its subordinate objects; Clock, NEQ, and Protocol Filter, and schedules a simulation Termination Event at the maximum simulation time. A *Start Simulation* message causes the Synchronizer object to request the next event which causes a transition to the Getting Next Event state. If the next event is the Termination event, the Synchronizer transitions to the End state. Otherwise, a *Request Safe To Proceed* message is sent to the Protocol Filter object and the Synchronizer remains in the Waiting for Response state until an answer is received.

If the response indicates *safe to proceed,* the event is sent to the Logical Process object to be passed to the appropriate object for processing. The Synchronizer then waits

to receive a *processing complete* message from the Logical Process object which results in the simulation time being updated and the next event on the queue being retrieved. If the response to the *request safe to proceed* message is false, then the next event is placed back on the NEQ, and a message is sent to the Node Manager to *check for a remote event*. Remote events are processed (placed on the queue) until the buffer is empty or a predefined number of remote events have been processed. When one of the two conditions is true the Synchronizer retrieves the next event from the NEQ and the loop repeats.

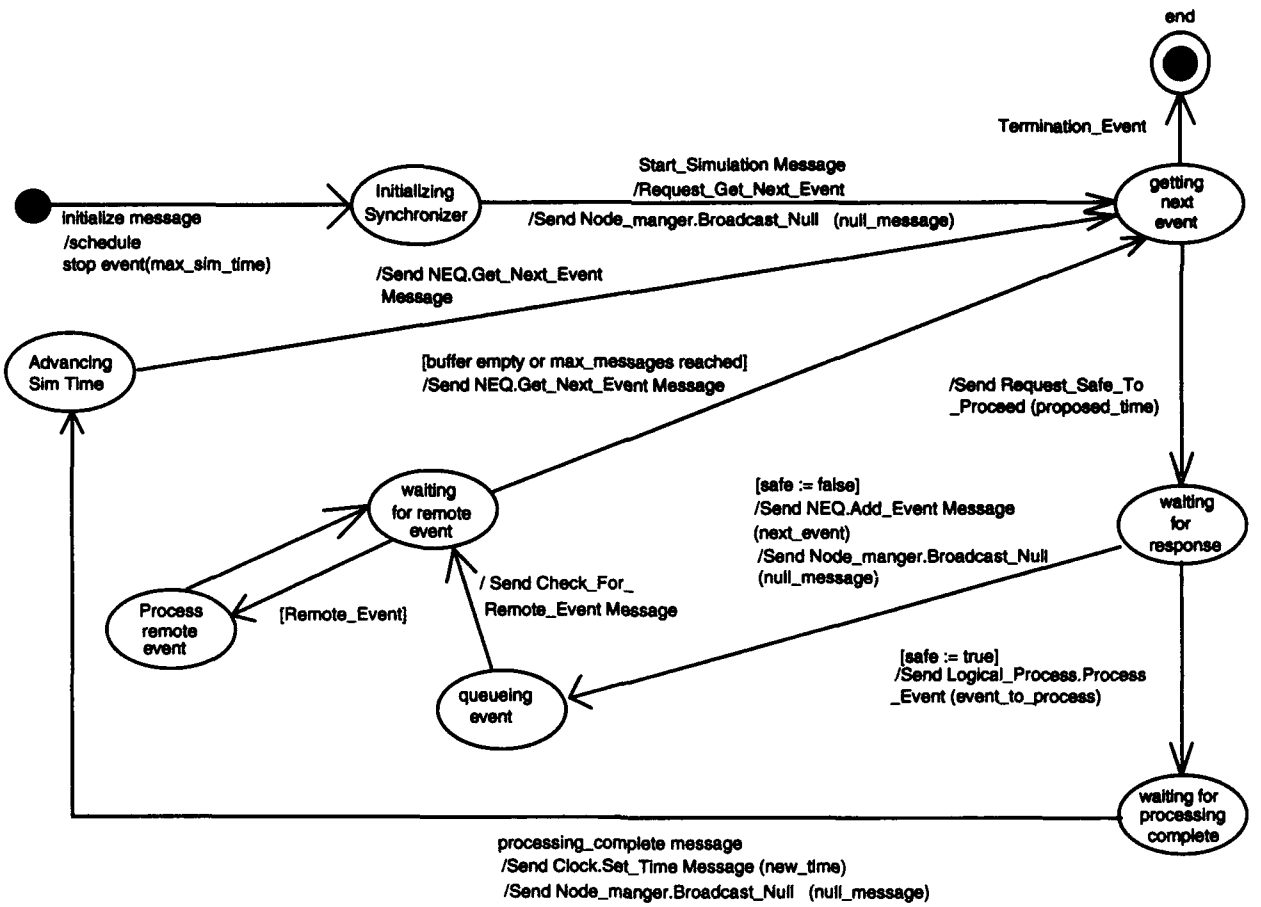


Figure 14. STD for the Synchronizer.

3.3.2.4 *STD for the Clock Object*

The STD for the Clock class is shown in Figure 15. During initialization the simulation time is set to zero and then the Clock object automatically transitions into the idle state. The Clock remains in the idle state until a message arrives to update, set, or get the simulation time. After the event is processed, a transition back to the idle state automatically occurs.

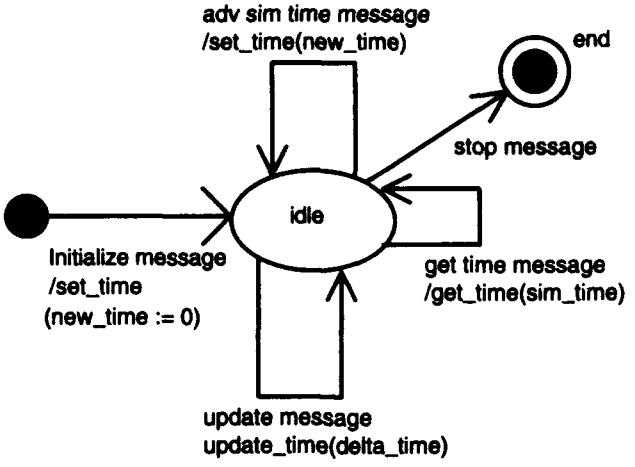


Figure 15. STD for Clock Class.

3.3.2.5 *STD for the Protocol Filter Object*

The STD for the Protocol Filter object is shown in Figure 16. During initialization, the desired time synchronization method is determined and the subclass for that method is created. Then, the Protocol Filter is either in the idle state or determining if it is safe to proceed with the next event.

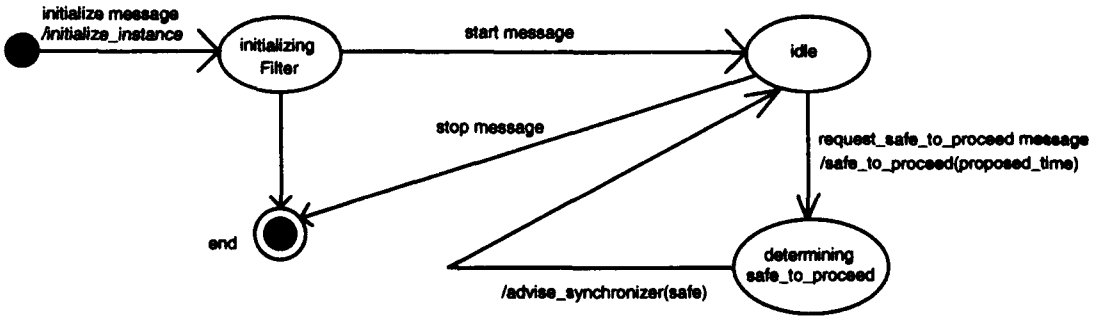


Figure 16. STD for Protocol Filter Class.

3.3.2.6 STD for the Logical Process Object

The State Transition Diagram for the Logical Process object is shown in Figure 17. An initialize message causes the Logical Process (LP) to enter the initialization state. After initialization is complete the LP transitions automatically to the idle state where it waits for an event to process. When the "pass event to object" message is received, the LP determines the location of the objects affected and passes the event to the affected objects in-turn. If the object is local it is passed directly, otherwise, it is sent to the Node Manager object to be included in a remote event message. While in the "passing event to object" state, if the object requests that the LP schedule a future event, the LP requests that it be placed on the local NEQ. When the event to process has been passed to all of the objects affected, a Processing Complete message is sent to the Synchronizer object relinquishing control and the LP transitions back to the idle state.

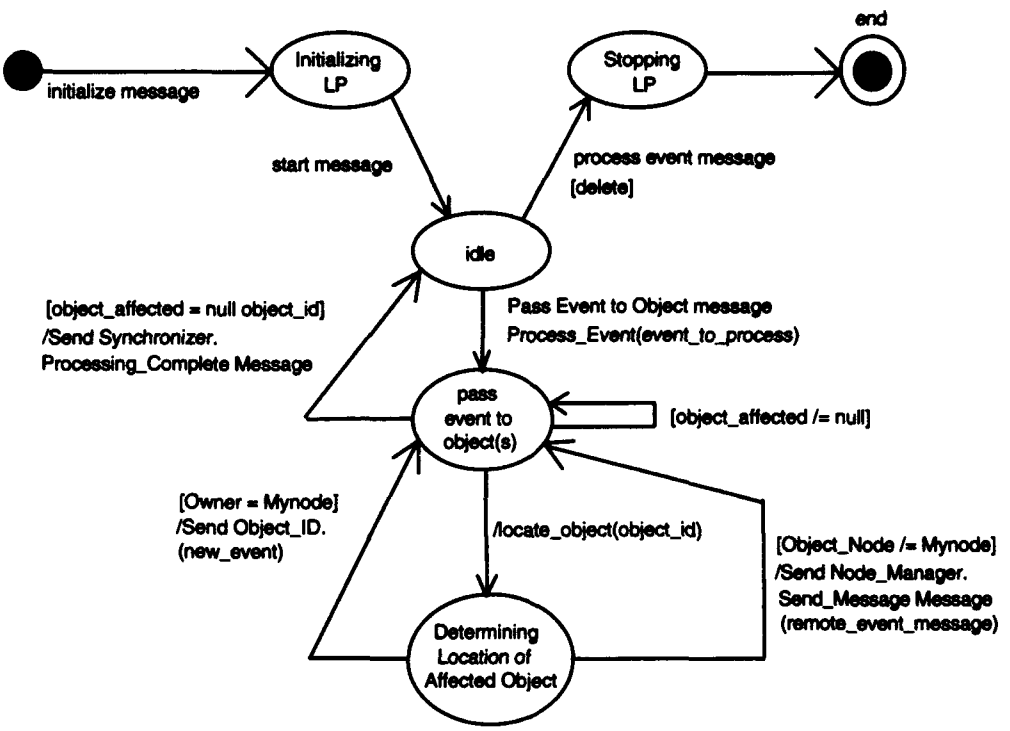


Figure 3-17. STD for the Logical Process (LP) Class.

3.3.2.7 STD for the Node Manager Object

The State Transition Diagram for the Node Manager object is shown if Figure 18. During initialization the Node Manger creates the appropriate subclass for the architecture that the simulation is to run on and then transitions to the idle state. If a stop message is received, the Node Manager broadcasts a message to the other nodes notifying them of the intent to terminate. Otherwise, when a *send remote message* (null or event type) message is received, the message is sent to the appropriate node by the subclass (Hypercube). If a *check for remote event* message is received, the Node Manager waits to receive a remote event type message from the subclass, and then passes the remote event to the Synchronizer object for processing and updates the appropriate input channel's time. If an add player type message is received, the player is passed to the Logical Process object to be added to the player map. After the remote event message has been passed to the appropriate object for processing, the Node Manager returns to the idle state.

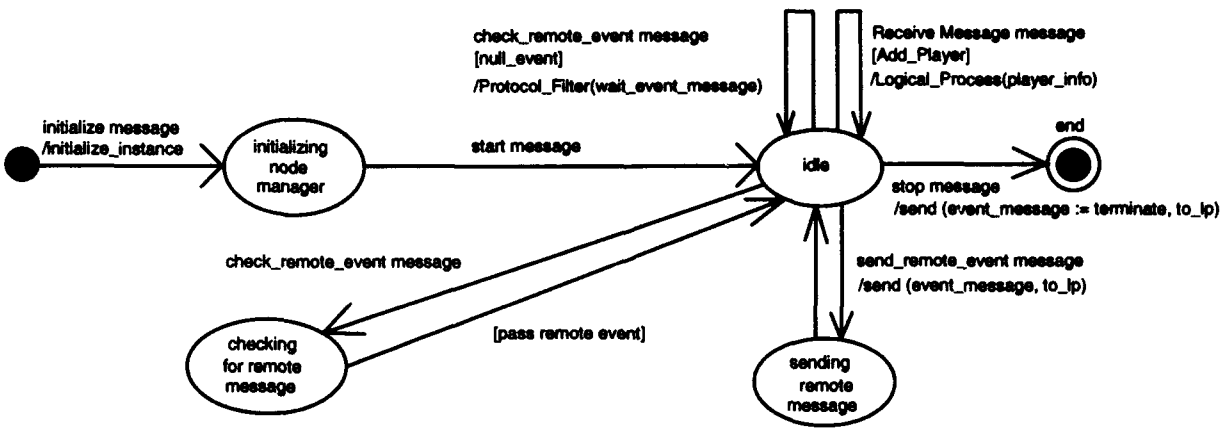


Figure 18. STD for Node Manager Class.

3.3.2.8 STD for the Player Object

The State Transition Diagram for the Player object is trivial. The Player object is either in an idle state or Processing an event state.

3.3.3 Functional Model.

The Functional Model for the Parallel Ada Simulation System is shown in Figure 19. Data flow between objects is indicated by the solid lines, and control flow is indicated by the dashed lines. A description of the data items can be found in the data dictionary contained in Appendix A.

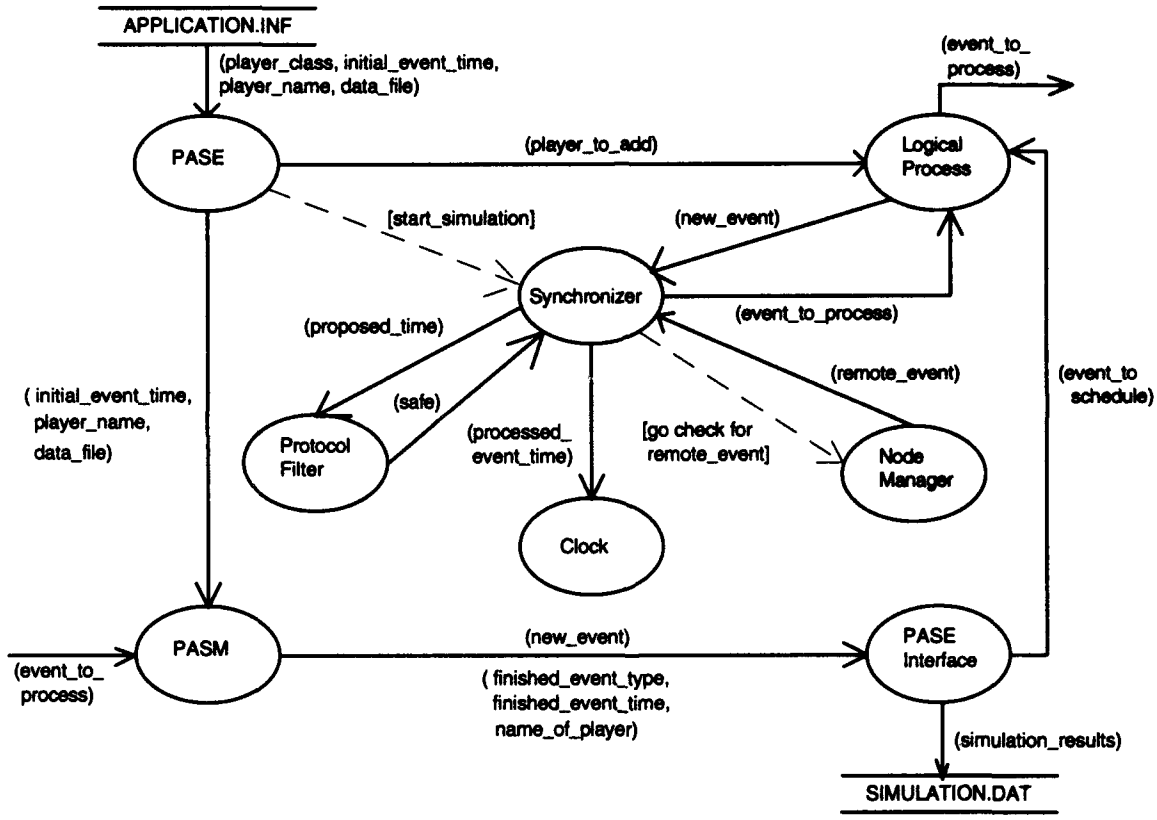


Figure 19. Functional Model for the Parallel Ada Simulation System (PASS).

IV. Design and Implementation

4.1 Object Oriented Design.

The Parallel Ada Simulation System (PASS) software is split into two major subsystems; the application, or Parallel Ada Simulation Model (PASM), and the Parallel Ada Simulation Environment (PASE). Section 4.1.1 provides an examination of the design details that apply to the PASM, and Section 4.1.2 provides the design for PASE.

4.1.1 Parallel Ada Simulation Model (PASM).

4.1.1.1 General Design Issues.

PASM was designed with modularity and the ease of building or changing the simulation scenario in mind. The following areas of PASM were structured in a fashion to provide the maximum flexibility in developing an application to run within the PASE.

4.1.1.1.1 Scenario Generation.

The application scenario information is read from a file (application.#) during the initialization state of the PASS. The suffix (#) of the application filename is determined during initialization, and appended to allow each node to read its own file. For example, node 1 would read file application.1, node 2 would read application.2, and so on. The executable is simply loaded on the desired number of nodes and if the files for each node exist, the simulation environment determines which nodes have been loaded and reads the corresponding application files.

The scenario is defined by providing following information:

Maximum_Simulation_Time The very first line in the application file defines the time at which the simulation will terminate. Although each node may terminate at different times, all nodes should be given the same termination time to prevent one node sending a message to another node which has terminated.

Player_Class The player class indicates the type of player to be instantiated, for instance Aircraft_Class.

Initial_Event_Time The initial event time defines the time that a player executes its first event, thus entering the simulation. An example for an aircraft instance would be a take-off event at 1000.0 Zulu.

Player_Name The player name provides the means for the modeler to identify with a particular player object. For instance, an aircraft can be given the identifier "EG1005", which could represent a tail number. The statement, "object_id 25 bombed object_id 32" is understood by the system but means little to the most modelers. However, "aircraft EG1005 drops a bomb on tank Bravo1" is much more meaningful.

Object_Data_File Each player has a data file associated with it. This data file is read during the initialization of each instance to initialize its parameters. In the case of a vehicle object, the data file name provided is that of the route.

The file structure makes it easy to change the application scenario and minimize communication costs. Players can be added to the simulation, vehicle routes may be changed, initial event times varied, and termination times defined by simply modifying the application file. In addition, the modeler can use *apriori* knowledge about object interaction to group players with maximum dependencies in the same file (assigned to the same node) to minimize the costs associated with passing messages. Each application file represents a "team" of players which are assigned to a logical process on a single node.

Route information for vehicles is contained in the route files. The points are provided within each file by listing the coordinates (x, y, z) sequentially from top to bottom with no spaces in between. During initialization of a vehicle object, this file is read and the vehicle's route created by first creating a route object and then creating a route_point object for each set of coordinate points and adding the object_id for the point to the route array. Each time a vehicle processes a route point event, a route point is retrieved from the route array and the coordinates for the point are queried.

4.1.1.1.2 Processing Events.

Each player class in the simulation has an event class associated with it. For example, AC_Event_Class is associated with the Aircraft_Class. The event types that Aircraft_Class must be able to process are indicated by the range of event types defined for AC_Event_Class. The range of events is provided by declaring Aircraft_Event_Type to be an Ada enumerated type. This provides a finite set of events that an Aircraft player must be able to process. For example,

```
type Aircraft_Event_Type is (Take_Off, Route_Point, Landing);

Aircraft_Event  :  Aircraft_Event_Type;
```

The Ada *case* construct was used within each player to handle various types of events.

```
case Aircraft_Event is
    when Take_Off =>
        statements
    when Route_Point =>
        statements
    when Landing =>
        statements
end case;
```

Events types may be added to a player by including them in the enumeration of events for the player and providing them as possible values of the *case* construct within the player code. The statements corresponding to the value within the *case* statement should be those required to process the event.

An event is generated for a player as follows:

- create an event object of the type associated with that player
- set the type of event to one in the range defined for that player type
- set the event time

- set up to three objects (by name) affected by the event
- schedule the event (place object_id on the NEQ)

At the preset simulation time, the event is taken off the NEQ and processed as follows:

- the object_id of the event is passed to the affected player object by the environment
- the object uses the event object_id to query the event type
- the case construct is used to execute the code which corresponds to the event type

All players are structured in the same way, which lends itself well to the generation and use of code templates for player objects.

4.1.1.1.3 Player Code Templates.

Like J-MASS, code templates are provided for class objects which may be easily modified for new types. Soft coded portions (those which are likely to change between Classes) are indicated by bold italics. This is convenient for classes such as the Player class where the types of events which need to be processed will change but the basic structure of a Player class remains the same. A player template can be used for a new class by defining the instance variables and methods unique to that subclass, and defining the *case* values as the possible event types for that player. The statements required to process each event type is then added for each event value. An example of a PASM template is provided in Appendix C.

4.1.1.2 Reused Code.

Existing code that was reused for PASM includes:

Location_Class Location class provides a method for describing the location of an entity in three dimensional space. This was used to create a Route Point subclass which describes the location of a vehicle's route points. It can also be used for a future enhancement to create an entity location subclass which can keep track of the location for each player in the simulation.

Root_Event_Class Root Event class defines a basic simulation event class. It is currently the super class for the Aircraft and Tank event subclasses. Root Event class can be used to create event subclasses for other types of players as well.

Route_Class Route class implements a class which maintains a set of coordinates which define a particular route. It is currently used to store vehicle routes.

4.1.1.3 Data Structures.

The only data structure implemented in PASM is used to store a vehicle's route in the route class. A queue is used to store the vehicle's route which allows access to the next route point in constant time. Route points may be added to either the front or back of the queue. Route points can be read destructively by invoking the method "Get_Next_Route_Point", or non-destructively by invoking the method "Query_Next_Route_Point".

4.1.2 Parallel Ada Simulation Environment (PASE).

4.1.2.1 General Design Issues.

The three characteristics of a system considered the most important for the PASE to possess were *modularity, modifiability,* and *portability.* First, the environment should be designed in a way which allows execution on more than one architecture without significant changes to the code (*portable*). Second, modifications to the configuration of the environment should be fairly simple to accomplish (*modifiable*); for example, changing from a conservative time synchronization approach to an aggressive. Finally, the characteristic of *modularity* contributes to the realization of the first two. PASE was designed to allow the user to "plug-in" various filters for time synchronization protocols and hardware interfaces for other parallel architectures. The following subsections describe the design and implementation features of PASE which achieve these characteristics.

4.1.2.1.1 Distributed Player Management.

To control communication between entities and allow a simulation to be distributed across *n*-nodes while remaining transparent to the user, a method for Distributed Player Management must be provided. For PASE this is accomplished by using a player map. The *Logical Processor* object on each node creates a map containing information on all players in the simulation. During the *Build Scenario* phase of the simulation, after a player is read from the application file, created, and initialized, it is sent to the *Logical Process* object to be added to the local map and broadcast to all other nodes to be added to their respective maps. After a node completes initialization, it broadcasts a signal to every other node in the simulation indicating that it is complete. This signals to the other nodes that no more remote players will be sent, allowing the simulation to start.

Frequent access to the map is required since it is accessed each time an event is taken off the *NEQ* to determine the owner (location) of the affected player. As such, a data structure is necessary that requires relatively small search and update times. Implementation of the data structure was accomplished using a map structure consisting of an array with 23 entries, each entry acting as a "bucket" which contains a set of player information records unique to that bucket. This map structure is refered to as an open hash table (2 : 212). The number 23 was chosen because it seemed reasonable for the number of players which may participate in a PASS simulation. In addition, a prime number is desired to minimize clustering. Clustering occurs when the same bucket is chosen more often than others. Each player's name is converted to an integer value which is used to store its player information record in the table (Figure 20) using the hashing function

bucket_id = integer value of player_name string mod 23

The player name was used since each name should be unique - if not, a multiple binding exception will be raised. Since each player name is unique, the conversion function returns a unique integer to be hashed, which prevents collisions. A collision occurs when the same

location is returned for two different players. Player object_Ids were not used since they may be the same if created on different nodes.

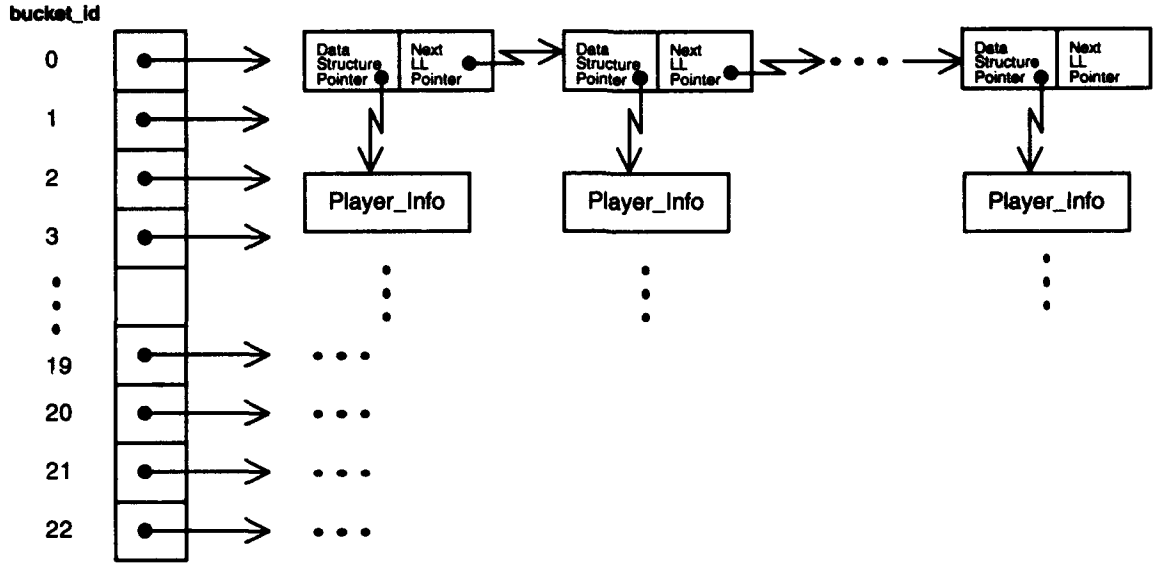


Figure 20. Structure of the PASE Logical Process TEAM map.

The time and space complexity of the map is a function of its extent (denoted by *n*), and the number of buckets (denoted by *b*). The space required by the map includes storage required for each node as well as for the array of buckets. Thus, the space complexity is on the order $O(b + n)$. Analysis of the time complexity demonstrates the clear advantages of hashing. The operations to store a player (bind) and retrieve a player (range of) both require that the map be searched for an occurrence of the domain. This search requires an order of $O(1 + n/b)$ time, assuming that the hashing function is accomplished in $O(1)$ and the player records are uniformly distributed.

Although migration of players between nodes is left for future research, this data structure will support it. A player can be moved from one node to another by deleting the player object on its current node, creating a new player object on the destination node, and modifying the owner field of the player's record in the map for each node.

4.1.2.1.2 Inter-Processor Communication.

Communication between *Logical Processes* on remote processors is accomplished by the Node Manager class and the Hypercube subclass. The Node Manager Class provides services such as maintaining the input/output channel times for each remote node, determining the minimum input channel time for all of the nodes, and providing the interface between the rest of the environment and the Hypercube object. The Hypercube provides the actual host architecture dependent routines that allow messages to be passed between processors. This modular structure allows PASS to be ported to a different architecture by replacing the Hypercube object with one which handles the communication protocol for the new host.

Distribution across *n*-nodes is supported since the node manager creates remote node objects dynamically. When an *Initialization Complete* message is received from a remote node, the Hypercube object notifies the Node Manager which creates a Remote Node object for that node. Identification of the remote node is accomplished using the *nodeinfo* command which returns the integer value of the node which sent the *Initialization Complete* message. This integer value is used to store the object_id for that node into an array of Remote Nodes. As messages are sent to or received from remote nodes, the integer value of the node is used to access the object_id in the array, which is then used to update the corresponding channel time for that node. When *n*-1 *Initialization Complete* messages have been received, a Remote Node object exists for each node in the simulation, and access is provided by using the node number to index the array of nodes. The number of nodes in the simulation (*n*) is determined by the dimension of the cube allocated. The minimum channel time is then determined by traversing the array and querying each Remote Node object for its current input/output channel time.

Requests to pass a remote event or null message to another node are passed to the Hypercube object by the Node Manager. Likewise, when the Hypercube object receives a message from a remote node, it is passed to the Node Manager to handle.

Communication between nodes on the Hypercube is accomplished using the three commands described in Section 3.2.2.3.1 (CPROBE, CSEND, CRECV). Each message has a particular type associated with it. CPROBE blocks until a message of the type indicated is waiting to be received. CSEND sends a message to another node and blocks (waits) until that message has been sent. CRECV initiates the receipt of a message and blocks the calling program until receipt of the message is complete. In each instance, blocking prevents the respective buffer from being overwritten before the message has been handled. Four types of messages are defined for the PASE.

Type 1 identifies a *Add Remote Player* Message,

Type 2 identifies an *Remote Initialization Complete* message,

Type 3 identifies a *Remote Event* message, and

Type 4 identifies a *Null* message.

A node can block for a message of a certain type, or if the message type is given the value -1, any message is accepted. The frequency at which remote messages are checked is an important consideration. If remote messages are not checked for often enough, a buffer overflow results. On the other hand, if too much time is spent receiving remote events instead of processing events, then an overflow of the NEQ occurs. After experimentation with the frequency of checking, and the quantity of messages received at each interval, a fair balance was determined. Each time that it was not safe to proceed with the next event (Proposed Time > Safetime), the process would block and wait for at least one message and accept up to a maximum of 25 messages.

4.1.2.1.3 Time Synchronization.

The basic Chandy-Misra conservative approach to time synchronization which was described in Section 2.3.1 is used to insure execution of events in proper time stamp order for the prototype implementation of the PASE. The Conservative Filter object is responsible for calculating the safetime which is the maximum time that a proposed event can be

scheduled to occur at and still be processed. Since the Chandy-Misra algorithm depends on the minimum input channel time, the Protocol Filter requests this time from the Node Manager. The Conservative Filter compares the proposed time with the minimum input channel time and sets safe equal to true if the proposed time is greater, or false otherwise.

Use of the Chandy-Misra approach requires that a method of deadlock avoidance be implemented. A deadlock state occurs when every processor is waiting to receive a message from another processor in order to advance its input channel time to something greater than the time of the event at the front of the NEQ (5 : 201). The easiest technique is to send out null messages - a message which conveys no real event but is simply used to advance the input channel time of the remote processors - whenever it is not safe to proceed. This approach to deadlock avoidance is expensive because the majority of messages transmitted are null messages which convey no useful event information. However, for the intent of this research, the null message approach was used as a baseline for future efforts.

To avoid deadlock, null messages are sent out at three separate times. During the simulation initialization phase, a null message is broadcast indicating the time of the initial event at the front of the NEQ. This allows each node to set the input channel times to an initial value. This allows at least one event to be processed, and more if two or more initial events occur at the same time. The second time a null message is sent is after an LP has finished processing an event. The time to which the simulation clock is advanced is broadcast to other nodes allowing their input channel times to be advanced. The third time a null message gets sent is when a proposed event time is greater than the safetime. A null message is broadcast with the time stamp of the next event on the queue to advance the other nodes' input channel time for that node to the time of the next event to processed. Although costly, broadcasting null messages at the times indicated assures execution of events in proper time stamp order. More efficient synchronization protocols are left for future research efforts.

4.1.2.2 Reused Code.

Existing code that was reused for PASE includes:

Clock_Class Clock class provides a method for implementing a clock object which maintains an LP's notion of the current simulation time. The time may be set to a new value, advanced by a given delta, or queried to get the current value.

NEQ_Class NEQ class defines a basic next event queue which is used to store an LP's simulation events in proper time stamp order.

4.1.2.3 Data Structures.

Two data structures are required for the Parallel Ada Simulation Environment. The first is used to implement the next event queue. The queue is implemented using a linear linked list structure. The event at the front of the queue is retrieved in constant time and an event is inserted on the queue on the order of $O(n)$, where n denotes the length of the queue. The queue has a space complexity of $O(The\_Size)$, where *The_Size* refers to the bounded size of the queue. The other data structure used is the player map described in Section 4.1.2.1.1.

V. Test Results

5.1 Introduction.

The initial testing on the simulation software was performed in a sequential environment on a SUN Sparc workstation and repeated on the host processor, the Intel IPSC/2 Hypercube. This initial testing was performed to determine if the simulation provided accurate results for a sequential Discrete Event Simulation (DES). The scenario used for the initial test is described in Section 5.2 and the results are discussed in Section 5.3. After successful completion of testing the sequential version of PASE, the simulation was configured to run in parallel with LPs interacting by passing messages between nodes. The scenario for the Parallel Discrete Event Simulation (PDES) test is described in Section 5.4 and the results are discussed in Section 5.5. The raw simulation output data for both the sequential test and the parallel test is provided in Appendix B.

5.2 Scenario for the Sequential Test.

The benchmark scenario was set up to test several aspects of the simulation system software in particular. First, it was necessary to see if the simulation environment was being created and initialized properly. Creation of the environment required that each class object get created, and each subclass get created, until every class in the environment was created. The initialization of the environment consisted of initializing instance variables to their appropriate initial values and the creation and scheduling of the simulation termination event at the given maximum simulation time.

Next it was necessary to determine if the "build scenario" procedure of the simulation was functioning properly. "Build scenario" is supposed to open the *application.inf* file, read each block of information belonging to a particular player, then create and initialize that player, for each player in the scenario. Initializing the player involves initializing the player's instance variables and using the initial event time read from the file to schedule the player's

initial event. Also, if the player was of type vehicle, the route file had to be read and the route loaded properly (all players in the baseline scenario are either vehicle type tank or aircraft). After the environment (PASE) and the application scenario (PASM) were created and initialized properly, the integrity of the actual execution of the discrete event simulation had to be tested.

To test the execution of the simulation it was necessary to insure that the scenario caused certain actions to occur. First, more then one type of event was desired to determine if the simulation could differentiate between the types and process them accordingly. Second, it was desired to have a number of players of different types to determine if the simulation could accept events generated by various players and pass them back to the correct player for processing. Also, it was desired to have a significant number of events occur at various times to test the simulation's ability to schedule and process events in the proper time stamp order.

Finally, it was necessary to see if the simulation terminated gracefully at the provided maximum simulation time, and whether or not the simulation output was generated to allow post simulation analysis of the data. Graceful termination can be described as running to completion, as opposed to terminating for some error.

The baseline scenario designed to test the above items is described in the *application.inf* file illustrated in Figure 21. The baseline consisted of ten separate players. All ten players were of the Vehicle type class. Half (5) of the players were Tank objects and the other half (5) were Aircraft objects. All ten players had different initial event times; engine start times for the tanks and takeoff times for the aircraft. In addition, all ten vehicles followed different routes. A sample Route file is contained in Figure 22. The route file consists of ten ordered route points with the x, y, and z coordinates provided in that order. Along with the player class, each player has a name associated with it that is used for identification purposes when the simulation output data is analyzed.

5.3 Sequential Test Results.

The results of the sequential simulation execution are contained in the simulation data file located in Section B.2 in Appendix B. Examination and analysis of the output data indicates by inference that everything functioned properly. All ten objects were created, processed their initial events at the prescribed times, followed their routes, fired their cannon or dropped a bomb if the route point fell within the range of any of the predefined target points, and landed or shut-off their engine at the final route point. The simulation terminated gracefully at the preset maximum simulation time and the simulation data was properly written to the output file. The results led to the conclusion that PASS functioned properly as a sequential DES.

```
* This file contains the application information for a
* PASE simulation scenario
* The file format is Player_Class (ie, Aircraft_Class)
*                    Initial_Event_Time (ie, 1000.0)
*                    Player_Name (ie, EG1001)
*                    Object_Data_File (ie, Route#)
* First object is an aircraft
Aircraft_Class
1000.0
EG1005
Route1
* Second object is a Tank
Tank_Class
997.0
Bravo1
Route6
* Third object is a Tank
Tank_Class
1001.0
Bravo2
Route7
* Forth object is an Aircraft
Aircraft_Class
1002.0
EG1125
Route2
*Fifth object is an Aircraft
Aircraft_Class
1002.5
EG1002
Route3
```

Figure 21. Application.inf file for the baseline test of the PASS.

```
* Sixth object is an Aircraft
Aircraft_Class
1003.0
EG1025
Route5
* Seventh object is a Tank
Tank_Class
990.0
Bravo3
Route8
* Eighth object is a Tank
Tank_Class
975.0
Bravo4
Route9
* Ninth object is an Aircraft
Aircraft_Class
1010.0
EG1010
Route4
* Tenth object is a Tank
Tank_Class
1005.0
Bravo5
Rout10
```

Figure 21 (cont.). Application.inf file for the baseline test of the PASS.

```
100.0
100.0
0.0
550.0
400.0
3.0
1001.0
401.0
4.0
1000.0
120.0
4.0
3500.0
500.0
4.0
5000.0
750.0
3.0
2501.0
601.0
2.0
2300.0
1500.0
2.0
3000.0
1750.0
3.0
3200.0
2200.0
3.0
```

Figure 22. Sample Route file for the baseline test of the PASS.

Application.0 (File for Node #0)

```
1100.0
* The very first line of this file must contain the
* maximum simulation time.
* This file contains the application information for a
* PASE simulation scenario
* The file format is Player_Class (ie, Aircraft_Class)
*              Initial_Event_Time (ie, 1000.0)
*              Player_Name (ie, EG1001)
*              Object_Data_File (ie, Route#)
*
* First object is an aircraft
Aircraft_Class
1002.0
EG1005
Route1
* Second object is a Tank
Tank_Class
998.0
Bravo1
Route8
* Third object is a Tank
Tank_Class
1012.0
Bravo2
Route9
* Fourth object is an Aircraft
Aircraft_Class
1003.0
EG1015
Route3
* Fifth object is a Tank
Tank_Class
1013.0
Bravo3
Route7
```

Application.1 (File for Node #1)

```
1100.0
* The very first line of this file must contain the
* maximum simulation time.
* This file contains the application information for a
* PASE simulation scenario
* The file format is Player_Class (ie, Aircraft_Class)
*              Initial_Event_Time (ie, 1000.0)
*              Player_Name (ie, EG1001)
*              Object_Data_File (ie, Route#)
*
* Sixth object is an aircraft
Aircraft_Class
1002.5
EG2005
Route3
* Seventh object is a Tank
Tank_Class
999.0
Bravo4
Route7
* Eighth object is a Tank
Tank_Class
1015.0
Bravo5
Route8
* Ninth object is an Aircraft
Aircraft_Class
1004.5
EG2015
Route3
* Tenth object is an Aircraft
Aircraft_Class
1001.2
EG2025
Route1
```

Figure 23. Application Files for Parallel Test.

5.3 Scenario for the Parallel Test.

The parallel test required that several new areas be examined. For the first parallel test, a total of ten players were loaded on two nodes (node 0, node 1). The scenario is described by the application files illustrated in Figure 23. To perform the tests in a parallel

environment several changes were made to the code to generate events for remote objects.

The following areas had to be tested for the parallel version:

(1) Distributed Player Management

a. Determine if a player object is local or remote.

b. Locate remote player objects.

c. Transmit remote events.

d. Receive remote events.

(2) Time Synchronization

(3) The ability to distribute the simulation across *n*-nodes.

To test the new areas, modifications were made to both the tank player object and the aircraft player object. The aircraft object was modified to generate a *Battle Damage* event for a tank object each time it dropped a bomb. A *Bomb Offset* of .025 simulation time units was used to separate the time that the bomb was released from the plane to the time impact was made with the tank. The aircraft object started with a quantity of eight bombs. The target that was scheduled for a *Battle Damage* event was determined by the number of bombs remaining on the aircraft. When eight bombs were left a *Battle Damage* event was scheduled for tank Bravo1, when seven bombs were left a *Battle Damage* event was scheduled for Bravo2, and so on. The tank object was modified by adding the *Battle Damage* event. The event simply involved incrementing a *Battle Damage Factor* each time the tank was bombed. Once the opportunity existed for one object to schedule an event for another object, the capability to test items 1 and 2 was accomplished.

The ability to distribute across *n*-nodes was tested by providing the required application files and loading the simulation on 1, 2, 4, and 8 nodes respectively. A total of 16 players (8 tanks and 8 aircraft) were used for all four configurations. This provided for anywhere from 16 players per node to 2 players per node. Ten different route files were used, each with ten route points (4 - 5 target points). Players were distributed across the

nodes to maximize inter-processor communication. This is not the preferred distribution for a normal simulation!

5.5 Parallel Test Results.

The results of the parallel simulation execution are contained in the simulation data file located in Section B.3 in Appendix B. Several items can be verified by analysis of the simulation data. First, all of the events in the list are in ascending timestamp order. Second, all of the players performed their initial events at the proper times. Looking at the more interesting items, is noted that when any aircraft dropped its first bomb a Battle Damage event was executed for Bravo1 approximately .025 time units latter. For instance, it is noted (in bold characters) when EG2025 dropped its first bomb at 1003.515 on node 1, .025 time units later at 1003.540 Bravo1 processed at battle damage event on node 0. In addition, when an aircraft EG2025 dropped its second bomb, a *Battle Damage* event was likewise scheduled for Bravo2 at .025 simulation time units later. Further inspection of the simulation results indicates that all events were handled correctly in both directions. The ability to locate remote players, send remote events, receive remote events, schedule remote events, and process remote events are all verified by inspection of the simulation data.

The ability to distribute the simulation across *n*-nodes tested equally well. The desired number of cubes was acquired using the *getcube* call and the *PASE_MAIN* program was loaded and executed automatically using the *load* command. Regardless of the number of nodes chosen, or the distribution of the players across them, the simulation would read the appropriate file for the node it was loaded on and run to completion writing the proper results to the simulation output file.

VI. Results, Conclusions, and Research Recommendations

6.1 Introduction.

This chapter summarizes the results of the work completed for this thesis effort. Section 6.2 looks at the tangible results accomplished as a result of the work performed. Section 6.3 states conclusions based on the results. Finally, Section 6.4 provides recommendations for further research in the area of Parallel Discrete Event Simulation (PDES).

6.2 Results.

The desired outcome of this thesis effort was met. The design, implementation, and test of a prototype Parallel Ada Simulation System to include both the application model and the environment in which the model could execute was successful. Specifically, the following resulted from this effort:

Parallel Ada Simulation Model (PASM) - The Parallel Ada Simulation Model describes the format of the application that runs in the Parallel Ada Simulation Environment (PASE) described below. As the name implies, the prototype application is written in the Classic Ada program design language. The object oriented design and implementation of PASM provides flexibility in the construction of simulation scenarios using existing components and the creation and integration of new components. The following PASM components were generated as a part of this thesis:

Player Class - Implements an abstract class which defines a Player object.

Vehicle Class - Describes a vehicle object which is a subclass of Player Class.

Aircraft Class - Describes a specialization of the Vehicle Class.

Tank Class - Describes a specialization of the Vehicle Class.

Root Event Class - Defines a basic simulation event class. (reused component)

Aircraft Event Class - Describes an Aircraft Event object which is a specialization of the Root Event Class.

Tank Event Class - Describes a Tank Event object which is a specialization of the Root Event Class.

Location Class - Defines an abstract 3D location class. (reused component)

Route Point Class - Specialization of the Location Class which defines the location of Route Points in 3 dimensions.

Route Class - Implements a class which contains a set of coordinates defining a route to be followed in sequential order. (reused component)

Parallel Ada Simulation Environment (PASE) - The Parallel Ada Simulation Environment provides the platform to execute models developed using the PASM structure. Other types of models may be executed in the PASE if an application interface is written which maps the interface requirements of the application to those of the PASE interface. The modular structure of the PASE allows changes to be made to the configuration of a simulation with little effort. The following PASE components were generated as a result of this thesis effort:

IO Manager Class - Implements an abstract class which provides the interface between the Parallel Ada Simulation System and any number of input/output devices.

Synchronizer Class - Implements a Class which is responsible for insuring that simulation events are executed in the proper time stamp order and that the simulation time is maintained.

Clock Class - Implements a Class which maintains and makes available the current simulation time. (reused component)

NEQ Class - Implements a class which provides a next event queue to store and retrieve simulation events in proper time stamp order. (reused component)

Protocol Filter Class - Implements an abstract class which is used to determine the safetime which indicates the latest time that an event to process may have and still be allowed to be processed.

Conservative Class - Implements a class which determines the safetime based on the basic Chandy-Misra conservative time synchronization protocol.

Logical Process Class - Implements a class which manages the player objects assigned to the local node and maintains the location of all simulation players.

Node Manager Class - Implements an abstract class which provides communication services between processor nodes on a parallel architecture.

Hypercube Class - Implements a specialization of the Node Manager class which provides communication services for the Intel iPSC/2 Hypercube.

6.3 Conclusions.

Several requirements for a Parallel Ada Simulation Executive were enumerated in Chapter 1 of this thesis. This research effort focused on developing a simulation environment that met or exceeded the listed requirements to improve the state of Modeling and Simulation Technology at AFIT, and in the DoD. The results of this research effort in accomplishing this are as follows:

1) **The Environment should be adaptable to the existing J-MASS and DIS standards.**

A thorough review of both the J-MASS and DIS documentation was performed in preparation for the design and coding of PASE. The ability to incorporate a PASS simulation

into the DIS, or execute a J-MASS model within the PASE certainly exists. Although further research in these areas would be required, PASE is structured in a way which would allow the specific interfaces to be incorporated. The *I/O Manager* is structured to allow interfaces to I/O devices to be added. One of the interface devices could well be a DIS Cell Adapter Unit. In addition, a J-MASS standard compatible model, or any other model for that matter, could be executed within the PASE if the corresponding application interface is developed.

2) The low- level machine interface should be transparent to the modeler.

This requirement is met by the modular design of the *Logical_Processor* and *Node_Manager* objects. The *Logical_Processor* determines the location of an object and if it's remote, initiates a remote event message. The *Node_Manager* passes messages to its subclass (*Hypercube* for the prototype) to be transmitted to the proper node. The modeler simply creates applications using the PASM format, generates the application files which contain the scenario information, and executes the simulation.

3) Parallel Simulation issues with regard to implementation of models on the ipsc/2 Hypercube and networked Sun SparcStations should be explored.

Research was performed in the area of existing parallel simulation environments currently in use at AFIT; namely SPECTRUM and TCHSIM. This research identified items about the existing environments which had a positive influence on successfully executing Discrete Event Simulations in parallel, and some of the problems that exist. In addition, a study was performed for CSCE 792, "*Parallel Architectures*", which studied the costs associated with communication and load imbalancing for the various types of multi-processor systems. The knowledge gained from this research was incorporated into the design of PASE. The result is a simulation environment that to this point performs as it was designed to with very little debugging.

4) The new executive must be robust enough to support both the conservative and optimistic paradigms, as well as sequential, parallel, or distributed operation.

The modular design of the *Protocol_Filter* object allows incorporation of any type of time synchronization filter into the environment. The *Protocol_Filter* will in-fact create the desired type of filter dynamically at runtime provided the corresponding filter object exists. The filter object is simply the algorithm required to implement the desired synchronization protocol to calculate the safetime. The PASS has been successfully tested in a sequential environment (Sun Sparc station), and in a parallel environment (Intel ipsc/2 Hypercube) in both sequential and parallel modes. Testing for distributed operation is beyond the scope of this research and left for further study.

5) The executive shall be developed using object-oriented techniques.

The use of object-oriented techniques was mandated to provide the characteristics desired for the new simulation environment which were discussed in Chapter 4 (Portability, Modifiability, Modularity). Rumbaugh's object-oriented modeling techniques were used for the analysis and design. The Classic Ada program design language was used to provide the full benefits of an object-oriented language. Ada is the language of choice for the DoD, and was chosen as the implementation language. The result, as demonstrated during integration and test, is a parallel discrete event simulation environment that possesses the desired characteristics and meets or exceeds the formulated requirements.

6.4 Recommendations for Further Study.

6.4.1 Time Synchronization Methods.

Currently a simple conservative filter has been written for the prototype implementation of the Parallel Ada Simulation Environment. This is an area where future efforts could increase the scope of experimentation with various time synchronization protocols. Filters implementing other conservative algorithms, optimistic algorithms, and

hybrid algorithms, can easily be inserted into the environment providing the opportunity to examine their effectiveness using a standard platform. (See template in Appendix C)

6.4.2 Interface to I/O Devices.

The I/O manager currently allows interface to the standard devices (monitor and keyboard), and external files. It is recommended that interfaces to VISIT and other entities such as DIS be implemented. The interface to VISIT should require minimal effort. A package can be developed using one of the existing interfaces as a template. The DIS interface would require more of an effort. The front end and back end of a cell adapter unit would have to be developed to translate incoming and outgoing PDUs.

6.4.3 Interface to J-MASS.

An application interface should be written which allows J-MASS models to be executed in the Parallel Ada Simulation Environment to provide flexibility and faster execution times.

6.4.4 Improvements to PASM.

Improvements can be made to the Parallel Ada Simulation Model to make it more in line with the sophistication of J-MASS models. This can be accomplished in at least two areas. First, components can be added to allow a larger selection of players and the level of detail at which objects are represented. In addition, a spatial manager can be written.

6.5 Summary.

The experience gained accomplishing the work involved in this thesis effort says much about the merit of object oriented design techniques and the Classic Ada program design language. Despite my lack of experience at programming and the fact that this effort was the first significant attempt at using Classic Ada for the implementation of a

system at AFIT, the desired outcome was met without major difficulties. I do not mean to trivialize the amount of work required. However, no problems were encountered that could not be resolved in a fair amount of time. Time spent during design definitely pays off during testing. In addition, the modular design allowed various components of PASE and PASM to be added or changed easily, providing flexibility in the configuration of simulations.

Appendix A. Data Dictionaries

A.1 PASE Object Model Data Dictionary.

procedure **PASE** is

-- *PASE* is the main procedure for the Parallel Ada Simulation Environment. Execution of *PASE* initializes the simulation, builds the simulation scenario, starts execution of the simulation, and stops the simulation when the current sim_time is equal to the value given max_sim_time.

procedure *Initialize_Simulation* ();

-- Initialize_Simulation causes PASE to transition to the Initializing Simulation state. In this state PASE is responsible for sending initialize messages to the Synchronizer, Logical_Process, Node_Manager and Output_Manager classes which causes each to transition into initialization states.

procedure *Build_Scenario;*

-- *Build_Scenario* reads the application file. Each player in the application file is created and initialized, and added to the local team map and broadcasted to other nodes in the simulation to be added to their maps.

procedure *Start_Simulation;*

-- Start_Simulation causes the first event on the NEQ to be retieved which causes a transition of the PASE to the execution state.

procedure *Stop_Simulation;*

-- If all events have been processed, or a termination event is encountered, *Stop_Simulation* sends a delete message to all of the objects in the simulation and the simulation is terminated.

CLASS **Output_Manager_Class** is

-- the *Out_Put_Manager* class accepts a request to output message item, or request for an input message item, and directs the request to the apprpriate subclass (File, Monitor, Visit, etc...) for processing.

INSTANCE METHOD Output_Data(Message_Item : in Message_Item_Type);
-- Directs Message_Item to the appropriate subclass for processing.

INSTANCE METHOD Input_Data(Message_Item : in Message_Item_Type);
-- Directs request for Message_Item to the appropriate subclass for processing.

end **Output_Manager_Class**;

CLASS **File_Class** is

INSTANCE METHOD *Create*(Filename : Filename_Type);
-- An instance method for creating an output file.

INSTANCE METHOD *Open*(Filename : Filename_Type);
-- An instance method for opening an output file.

INSTANCE METHOD *Read*(Filename : Filename_Type);
-- An instance method for reading an output file.

INSTANCE METHOD *Write*(Filename : Filename_Type);
-- An instance method for writing to an output file.

INSTANCE METHOD *Close*(Filename : Filename_Type);
-- An instance method for closing an output file.

end **File_Class**;

class **Synchronizer_Class** is

-- The class **Synchronizer_Class** is responsible for all time sychronization activities; that is ensuring that events are executed in the proper time stamp order. The Synchronizer_Class is an aggregate of the Clock_Class, the NEQ_Class, and the Protocol_Filter_Class.

INSTANCE METHOD *Start_Simulation*;
-- *Start_Simulation* invokes NEQ.Get_Next_Event which gets the first event in the NEQ and starts the simulation process. In addition, the time of the first event on the queue is broadcast to all other nodes in the simulation to allow them to initialize their input channel times.

INSTANCE METHOD *Request_Get_Next_Event*;
-- *Request_Get_Next_Event* sends a message to the NEQ object to get the event at the front of the NEQ. If the event is a *Stop_Event*, a null message is broadcast to the other nodes to advance thier input channel times and the simulation terminates. If the event is not a *Stop_Event*, the time of the next eventis retrieved and *safe_to_proceed* message is sent to the Protocol_Filter_Object..

INSTANCE METHOD Send_Request_For_Safe_To_Proceed
(proposed_time : out Simulation_Time_Type,
safe : in boolean);

-- An instance method which generates a *request_safe_to_proceed* message, which causes the appropriate protocol filter to determine if it is *safe* to proceed with the next event given the *proposed time* of the event.

INSTANCE METHOD Go_Check_For_Remote_Event;

-- *Go_Check_For_Remote_Event* generates a *check_for_remote_event* message, which causes the node manager to block until either a remote event or null message is recieved.

INSTANCE METHOD *Process_Remote_Event*
(New_Remote_Event : in Object_ID);

-- *Process_Remote_Event* processes remote events as they are recieved from the Node_Manager object by creating local events and placing them on the NEQ.

INSTANCE METHOD Queue_New_Event (generated_event : in Object_ID);

-- *Queue_New_Event* generates an *add_event* message for the NEQ, which causes the event generated from a remote LP to be placed in time stamp order on the queue.

INSTANCE METHOD *Processing_Complete;*

-- *Processing complete* sends an adv_sim_time message, which causes the clock object to advance the simulation time to the new time. Then a message is sent to the NEQ object to *Get_The_Next_Event,* and the processing starts over.

end **Synchronizer_Class;**

class **Clock_Class** is

-- The clock class provides the mechanism for a logical process to maintain its notion of simulation time and aids in the time synchronization process.

INSTANCE METHOD *Set_Time (new_time* : in float);
-- An instance method to set the clock to a specified time.

INSTANCE METHOD *Get_Time* (Sim_Time : out float);
-- An instance method which returns the current simulation time.

INSTANCE METHOD *Update_Time* (Delta_Time : in float);
-- An instance method which adds a delta to the current simulation time.

end **Clock_Class;**

CLASS **Next_Event_Queue_Class** is

-- The class **Next_Event_Queue_Class** provides the system with a queue in which events are maintained in time stamped order. Items are placed on the queue, and the system keeps track of a pointer to the event. When the pointer (Object_ID) is returned for the event at the front of the queue, the system stops tracking it.

INSTANCE METHOD *Add_Event*(New_Event : in Object_Id);
-- An instance method for adding an event to the NEQ in time stamp order.

INSTANCE METHOD *Get_Event* (New_Event : out Object_Id);
-- An instance method which returns the first event from the NEQ.

INSTANCE METHOD *Size* (Queue_Length : out Natural);
-- An instance method which returns the number of events currently on the NEQ.

INSTANCE METHOD *Next_Event_Time* (Event_Time : out float);
-- An instance method which returns the time stamp of the next event on the NEQ.

INSTANCE METHOD *Simultaneous_Events* (Number_Of_Events : out Natural);
-- An instance method which returns the number of events at the head of the NEQ with the same time stamp.

INSTANCE METHOD *Show_Next_Event_Queue*;
-- An instance method for displaying the current contents of the queue.

INSTANCE METHOD *Clear*;
-- An instance method for clearing the NEQ.

end **Next_Event_Queue_Class**;

class **Protocol_Filter_Class** is

-- The class **Protocol_Filter_Class** is responsible for implementing the chosen time synchronization protocol indicated in the application information file. This can be instantiated for any given time sychronization protocol by adding the instance methods to a new instance which implement the desired protocol. The Protocol_Filter_Class calculates a Safe_Time based on the protocol method and returns a boolean to the synchronizer indicating whether it is safe_to_proceed with an event given its proposed time stamp.

INSTANCE METHOD *Safe_To_Proceed*(Proposed_Time : in Simulation_Time_Type,
Safe : out boolean);

-- An instance method which determines whether it is safe to send an event to the appropriate object for processing based on the safetime and the event's time stamp.

end **Protocol_Filter_Class**;

class **Logical_Process_Class** is

-- The class **Logical_Process_Class** is responsible for management of the objects assigned to the local node. The Logical_Process_Class maintains a map of all players in the simulation which contains the owner of the player, the player's object_id, and the player's class. When a local object generates an event, the LP class sends a *Schedule Event* message to the Synchronizer object and the event is placed on the NEQ. When an event is recieved for processing, the Logical_Process_Class determines whether the affected object is local or remote by checking the player map, then passes the event to the player for processing if it is local or sends the event to the node manager if the object is remote.

METHOD *Pass_Event_To_Object* (Event_To_Process : in Object_Id);
-- *Pass_Event_To_Object* determines which events are effected by Event_To_Process and passes the event to each object in-turn for processing. When all objects that are affected have processed the event, a *Processing_Complete* message is sent to the Synchronizer Class..

INSTANCE METHOD *Schedule_Event* (Event_To_Schedule : in Object_ID);
-- *Schedule_Event* sends an event to the Synchronizer class to be placed on the NEQ..

INSTANCE METHOD *Add_Player* (Player_To_Add : in Player_Info_Type);
-- *Add_Player* uses the player's name to hash a key value then uses that key to add the player to the nodes player map.

end **Logical_Process_Class**;

class **Node_Manager_Class** is

-- The class **Node_Manager_Class** is responsible for all communication activity between nodes on the host parallel architecture. Although for the prototype environment this will be instantiated to provide communication services on the Intel IPSC/2 hypercube, it could very well be instantiated for any architecture. The Node_Manager uses the subclass (Hypercube for this implementation) to process communication requests. It is more or less an interface between the rest of the environment and its subclass.

INSTANCE METHOD Send_Event_Message(Process : Node_Type,
New_Event : Remote_Event_Type);
-- *Send_Event_Message* passes New_Event to the Hypercube subclass to be sent to the Process that the object which it affects is owned by.

INSTANCE METHOD *Send_Null_Message* (Null_Message_Out : Null_Message_Type);
-- Sends a request to the Hypercube subclass to broadcast a null message to all other nodes in the simulation.

INSTANCE METHOD *Check_For_Remote_Event*;

-- Sends a request to the Hypercube subclass to block until a remote event or null message is recieved from a remote node.

INSTANCE METHOD Create_*Remote_Node* (From_Node : Node_Type);
-- Creates a remote node object every time an *initializtion complete* message is recieved from a node (From_Node).

INSTANCE METHOD *Remote_Update* (New_Null_Message : Null_Message_Type);
-- Sends a message to the Remote_Node object to update its input channel time.

INSTANCE METHOD *Minimum_Input_Channel_Time* (Minimum_Time : Sim_Time_Type);
-- Determines the minimum input channel time (used by the Protocol_Filter class).

INSTANCE METHOD *Pass_Remote_Event* (Remote_Event_To_Pass : Remote_Event_Type);
-- Passes a remote event to the Logical Process class for processing.

end **Node_Manager_Class**;

CLASS **Remote_Node_Class** is

-- the *Remote_Node* class provides the means to maintain the input and output channel times for other nodes in the simulation.

INSTANCE METHOD Set_Channel_Time(Channel_Time : in Simulation_Time_Type_Type);
-- Sets either the input or output channel time for a given node.

INSTANCE METHOD Query_Channel_Time(Channel_Time : in Simulation_Time_Type_Type);
-- Returns either the input or output channel time for a given node.

end **Remote_Node_Class;**

A.2 PASE Dynamic Model Data Dictionary.

class **Synchronizer_Class** is

-- The class **Synchronizer_Class** is responsible for all time sychronization activities; that is ensuring that events are executed in the proper time stamp order. The Synchronizer_Class is an aggregate of the Clock_Class, the NEQ_Class, and the Protocol_Filter_Class.

-- Initialize message received

-- Causes a transition to the initializing Synchronizer state.

-- Start_Simulation message received

-- cuases transition to the getting next event state
-- start message invokes

INSTANCE METHOD Request_Get_Next_Event (event_id : in event_id_type);
-- Request_Get_Next_Event sends a get_next_event message to the NEQ object which in-turn generates a next_event_message if an event exists and returns the object_id of the event at the front of the neq.

-- Next_Event received

-- cuases transition to the waiting for response state
-- next_event_message invokes

INSTANCE METHOD Send_Request_For_Safe_To_Proceed
-- *Send_Request_For_Safe_To_Proceed* sends a request_for_safe_to_proceed message to the Protocol_Filter object which determines whether it is safe to proceed with the current event and in-turn generates a safe_to_proceed message with a boolean value assigned to the parameter *safe*.

-- Safe_To_Proceed message received

-- cuases transition to the waiting for processing complete state if *safe* = true
-- or the queueing event state if *safe* = false
-- safe_to_proceed message invokes

-- when *safe* = true
INSTANCE METHOD Logical_Process.Pass_Event_To_Object;

-- Event_To_Process is passed to the Logical_Process object for processing.

-- when *safe* = false
INSTANCE METHOD Request_Add_Event (premature_event : out type_of_event);

-- Request_Add_Event generates an add_event message for the NEQ object passing as a method parameter the event to be requeued which causes the Synchronizer to enter the queueing next event state.

INSTANCE METHOD Node_Manager. Send_Null_Event;

-- Causes a null message to be broadcast to all other nodes.

-- automatic transition from the queueing event state to the waiting for remote event state occurs after the add_event message has been sent

-- the transition invokes

INSTANCE METHOD Check_For_Remote_Event;

-- Check_For_Remote_Event generates a check_remote_event message for the Node_Manager object, which causes the node manager to receive any remote event messages or null event messages for processing. The Node_Manager continues to recieve messages until the buffer is empty, or the maximum number allowed has been reached. When control is returned from the Node Manager, the synchronizer transitions automatically to the getting next event state.

-- Processing_Complete message received

-- causes transition from the waiting for processing
-- complete state to the advancing sim time state
-- processing_complete message invokes

INSTANCE METHOD Node_Manager.Send_Null_Message;

-- Causes a null message with the processed events time to be broadcast to other nodes.

INSTANCE METHOD Send_Adv_Sim_Time_Msg(new_time : out float);

-- Send_Adv_Sim_Time_Msg generates an adv_sim_time message for the Clock object passing as a method parameter the new_time which is equal to the event time of the event just processed. The adv_sim_time message causes the clock object to advance the simulation time to the new time. Aftre the adv_sim_time_msg has been sent, the synchronizer requests_get_next_event and automatically transitions to the get next event state.

end **Synchronizer_Class**;

class **Logical_Process_Class** is

-- The class **Logical_Process_Class** is responsible for management of the objects assigned to the local node. The Logical_Process_Class maintains a map of all players in the simulation which contains the owner of the player, the player's object_id, and the player's class. When a local object generates an event, the LP class sends a *Schedule Event* message to the Synchronizer object and the event is placed on the NEQ. When an event is recieved for processing, the Logical_Process_Class determines whether the affected object is local or

remote by checking the player map, then passes the event to the player for processing if it is local or sends the event to the node manager if the object is remote.

-- Initialize message received

-- Causes a transition to the Initializing state.

-- Start message received

-- cuases transition to the Idle state

-- Pass_Event_To_Object message received

-- if event.type = stop_event

-- causes transition to the Stopping LP state
-- invokes

INSTANCE METHOD Stop_Object(object_id : object_id_type);

-- Stop_Object generates stop messages for all of the objects assigned to this Logical Process which stops the object tasks if they are implemented as tasks then transition to end Logical Process occurs.

-- if event.type ≠ stop_event

-- causes transition to the passing event to objects state

-- Location of object is determined

-- if the object is remote

-- a Node_Manager.Send_Remote_Event message is generated
-- Node Manager passes the event to the appropriate node for processing

-- else if the object is local

--invokes

INSTANCE METHOD Request_Process_Event (event_to_process : object_id_type);

-- Request_Process_Event generates a process_event message for the appropriate player object and passes it the event for processing. A transition from this state occurs after all affeccted objects have recieved and processed the event to pass.

-- Scedule_Event message received
-- while in the passing event to objects state

-- Schedule_Event message invokes

INSTANCE METHOD Synchronizer.Queue_New_Event;

-- Queue_New_Event places the New Event on the NEQ.

-- automatic transition back to the pass event to objects state occurs after the Queue_New_Event message has been sent

-- object_affected is null or three objects processed

-- causes transition to the idle state
-- invokes

INSTANCE METHOD Synchronizer.Processing_Complete;

-- automatic transition back to the idle state occurs after the processing_complete message has been sent.

end **Logical_Process_Class**;

class **Node_Manager_Class** is

-- The class **Node_Manager_Class** is responsible for all communication activity between nodes on the host parallel architecture. Although for the prototype environment this will be instantiated to provide communication services on the Intel iPSC/2 hypercube, it could very well be instantiated for any architecture.

-- Initialize message received

-- Causes transition to the Initializing Node Manager state.

-- Start message received

-- cuases transition to the idle state

-- Send_Message message received

-- causes transition to the sending remote message state
-- invokes

-- depending on the type of message, either

INSTANCE METHOD Send_Event_Message;
or
INSTANCE METHOD Send_Null_Message;

-- Send_Message sends the event message to the remote node where the
-- destination process is located.
-- transition back to the idle state is automatic.

-- Go_Check_For_Remote_Event message is received

-- causes transition to the checking for remote message state
-- go_check_for_remote_event message invokes

INSTANCE METHOD Hypercube.Wait_For_Remote_Message;

-- Node Manager remains in the waiting for remote message state, until control is passes back from the Hypercube object (all remote messages have been recieved or the maximum number have been processed).

-- Pass_Remote_Event message is received

-- causes the Node_Manager to pass the remote event to the Synchronizer to be processed.

-- Add_Player message is received

-- causes the Node_Manager to pass the New Player to the Logical Process object
-- to be added to the local map.

-- Stop_Event message is received

-- causes transition to the end state

Node_Manager_Class;

Note: The Object Model Data Dictionary entries adequately describe the relatively simple behavior of the Clock Class, Output_Manger Class, and Protocol Filter class. No Dynamic Model data dictionary entries are provided for these objects.

A.3 Functional Model Data Dictionary.

APPLICATION.INF - The application information file which contains the scenarion information for a given node.

data_file - identifies the name of the route file for a particular vehicle object.

event_to_process - object_id of the next event to be processed.

event_to_schedule - object_id of an event passed from a player to the logical processor for scheduling.

finished_event_time - the time of the last event processed.

finished_event_type - the type of the last event processed

go_check_for_remote_event - control signal which causes the Node Manager to block until remote message have been recieved and processed.

initial_event_time - the time at which a player schedules its initial event.

new_event - object of a new event generated by a player and passes to the PASE_INTERFACE to foward to the LP manager.

player_class - identifies the class of a player (i.e., Aircraft_Class or Tank_Class).

player_name - identifies a player's name (i.e., Bravo1 or EG1005).

player_to_add - is a player info record which is passed to the LP manager to be added to the local map.

processed_event_time - the time of the last event, used to update the simulation clock.

proposed_time - represents the time of the event at the front of the queue which is being considered for processing.

remote_event - an event generated and received from a remote node for a local player.

safe - a boolean value indicating whether or not it is safe tp proceed with the proposed event.

SIMULATION.DAT - The output file that the simulation results are written to.

simulation_results - a record containing the simulation information which is written to the simulation results file.

start_simulation - a control signal which causes the first event to be retrieved from the NEQ starting the simulation.

Appendix B. Simulation Output

B.1. Introduction.

The following pages contain the raw output from the execution of the Parallel Ada Simulation System base scenario described in Chapter 5 of this thesis. The output of the sequential test are provided in Section B.2, and those for the parallel test are contained in Section B.3. When an object completes processing of an event it sends a message to the PASE_INTERFACE object to write the simulation information to the output file simulation.dat. The information contained in the output is the Player Name in the first column, the type of event that was processed in the second column, and the simulation time of the event in the final column. See Chapter 5 for a disscussion of the results.

B.2 Simulation Output for Sequential Test

**********PASE Simulation Results**********

Player's ID	Event Type	New Simulation Time
Bravo4	START_ENGINE_EVENT	975.000
Bravo4	ROUTE_POINT_EVENT	976.763
Bravo4	ROUTE_POINT_EVENT	982.718
Bravo4	FIRE_CANNON	982.720
Bravo4	ROUTE_POINT_EVENT	988.841
Bravo3	START_ENGINE_EVENT	990.000
Bravo3	ROUTE_POINT_EVENT	990.619
Bravo4	ROUTE_POINT_EVENT	995.387
Bravo3	ROUTE_POINT_EVENT	996.030
Bravo1	START_ENGINE_EVENT	997.000
Bravo1	ROUTE_POINT_EVENT	997.891
EG1005	TAKE_OFF_EVENT	1000.000
EG1005	ROUTE_POINT_EVENT	1000.772
Bravo2	START_ENGINE_EVENT	1001.000
Bravo2	ROUTE_POINT_EVENT	1001.619
Bravo3	ROUTE_POINT_EVENT	1001.870
Bravo3	FIRE_CANNON	1001.872
EG1125	TAKE_OFF_EVENT	1002.000
EG1005	ROUTE_POINT_EVENT	1002.313
EG1005	DROP_BOMB	1002.314
EG1002	TAKE_OFF_EVENT	1002.500
Bravo4	ROUTE_POINT_EVENT	1002.616
EG1125	ROUTE_POINT_EVENT	1002.939
EG1025	TAKE_OFF_EVENT	1003.000
Bravo1	ROUTE_POINT_EVENT	1003.535
Bravo1	FIRE_CANNON	1003.537
EG1005	ROUTE_POINT_EVENT	1003.751
EG1002	ROUTE_POINT_EVENT	1003.859
EG1002	DROP_BOMB	1003.861
EG1025	ROUTE_POINT_EVENT	1004.360
EG1125	ROUTE_POINT_EVENT	1004.611
Bravo5	START_ENGINE_EVENT	1005.000
EG1025	ROUTE_POINT_EVENT	1005.905
EG1002	ROUTE_POINT_EVENT	1005.983
Bravo5	ROUTE_POINT_EVENT	1006.643
Bravo5	FIRE_CANNON	1006.645
EG1125	ROUTE_POINT_EVENT	1006.896
EG1125	DROP_BOMB	1006.898
Bravo2	ROUTE_POINT_EVENT	1007.030
EG1025	ROUTE_POINT_EVENT	1008.050
Bravo3	ROUTE_POINT_EVENT	1008.271
EG1005	ROUTE_POINT_EVENT	1008.802
EG1002	ROUTE_POINT_EVENT	1008.946
Bravo1	ROUTE_POINT_EVENT	1009.375
Bravo1	FIRE_CANNON	1009.377
Bravo4	ROUTE_POINT_EVENT	1010.317

**********PASE Simulation Results**********

Player's ID	Event Type	New Simulation Time
EG1125	ROUTE_POINT_EVENT	1009.551
EG1010	TAKE_OFF_EVENT	1010.000
EG1010	ROUTE_POINT_EVENT	1011.183
EG1025	ROUTE_POINT_EVENT	1012.215
EG1002	ROUTE_POINT_EVENT	1012.267
Bravo5	ROUTE_POINT_EVENT	1012.477
Bravo2	ROUTE_POINT_EVENT	1012.935
Bravo2	FIRE_CANNON	1012.937
EG1010	ROUTE_POINT_EVENT	1012.969
EG1125	ROUTE_POINT_EVENT	1013.003
Bravo3	ROUTE_POINT_EVENT	1015.169
EG1010	ROUTE_POINT_EVENT	1015.286
Bravo1	ROUTE_POINT_EVENT	1015.921
EG1005	ROUTE_POINT_EVENT	1016.024
EG1002	ROUTE_POINT_EVENT	1016.113
EG1125	ROUTE_POINT_EVENT	1016.905
EG1025	ROUTE_POINT_EVENT	1017.072
Bravo4	ROUTE_POINT_EVENT	1018.740
Bravo4	FIRE_CANNON	1018.742
Bravo5	ROUTE_POINT_EVENT	1018.960
Bravo2	ROUTE_POINT_EVENT	1019.335
EG1005	ROUTE_POINT_EVENT	1019.699
EG1005	DROP_BOMB	1019.701
EG1010	ROUTE_POINT_EVENT	1019.904
EG1010	DROP_BOMB	1019.906
EG1002	ROUTE_POINT_EVENT	1020.354
EG1125	ROUTE_POINT_EVENT	1021.569
Bravo3	ROUTE_POINT_EVENT	1022.305
Bravo1	ROUTE_POINT_EVENT	1022.994
EG1025	ROUTE_POINT_EVENT	1023.334
EG1005	ROUTE_POINT_EVENT	1023.622
EG1002	ROUTE_POINT_EVENT	1025.229
Bravo5	ROUTE_POINT_EVENT	1025.967
EG1125	ROUTE_POINT_EVENT	1026.190
EG1125	DROP_BOMB	1026.192
Bravo2	ROUTE_POINT_EVENT	1026.207
EG1010	ROUTE_POINT_EVENT	1027.133
EG1010	DROP_BOMB	1027.135
Bravo4	ROUTE_POINT_EVENT	1027.605
EG1005	ROUTE_POINT_EVENT	1028.584
Bravo3	ROUTE_POINT_EVENT	1029.731
EG1002	ROUTE_POINT_EVENT	1029.851
EG1002	DROP_BOMB	1029.853
Bravo1	ROUTE_POINT_EVENT	1030.560
EG1025	ROUTE_POINT_EVENT	1030.759
EG1125	LANDED	1031.048
EG1002	LANDED	1034.964

**********PASE Simulation Results**********

Player's ID	Event Type	New Simulation Time
Bravo5	ROUTE_POINT_EVENT	1033.148
Bravo2	ROUTE_POINT_EVENT	1033.846
EG1005	LANDED	1034.132
Bravo4	STOP_ENGINE	1036.725
EG1010	ROUTE_POINT_EVENT	1037.881
Bravo3	ROUTE_POINT_EVENT	1037.938
Bravo3	FIRE_CANNON	1037.940
Bravo1	ROUTE_POINT_EVENT	1038.767
Bravo1	FIRE_CANNON	1038.769
EG1025	ROUTE_POINT_EVENT	1040.023
Bravo5	ROUTE_POINT_EVENT	1041.521
Bravo5	FIRE_CANNON	1041.523
Bravo2	ROUTE_POINT_EVENT	1042.254
Bravo3	STOP_ENGINE	1046.788
Bravo1	ROUTE_POINT_EVENT	1047.878
EG1010	ROUTE_POINT_EVENT	1049.930
EG1025	LANDED	1049.991
Bravo5	ROUTE_POINT_EVENT	1050.871
Bravo2	ROUTE_POINT_EVENT	1051.697
Bravo1	STOP_ENGINE	1057.214
Bravo5	ROUTE_POINT_EVENT	1060.090
Bravo2	STOP_ENGINE	1061.364
EG1010	ROUTE_POINT_EVENT	1063.027
Bravo5	STOP_ENGINE	1069.707
EG1010	LANDED	1076.541
All_Players	SIMULATION_END_EVENT	1100.000

****

B.3 Simulation Output for Parallel Test.

Node 0

**********PASE Simulation Results**********

Player's ID	Event Type	New Simulation Time
Bravo1	START_ENGINE_EVENT	998.000
Bravo1	ROUTE_POINT_EVENT	998.619
EG1005	TAKE_OFF_EVENT	1002.000
EG1005	ROUTE_POINT_EVENT	1002.772
EG1015	TAKE_OFF_EVENT	1003.000
Bravo1	**BATTLE_DAMAGE**	**1003.540**
Bravo1	BATTLE_DAMAGE	1003.887
Bravo1	ROUTE_POINT_EVENT	1004.030
EG1005	ROUTE_POINT_EVENT	1004.313
EG1005	DROP_BOMB	1004.314
Bravo1	BATTLE_DAMAGE	1004.340
EG1015	ROUTE_POINT_EVENT	1004.359
EG1015	DROP_BOMB	1004.361
Bravo1	BATTLE_DAMAGE	1004.387
Bravo2	**BATTLE_DAMAGE**	**1005.085**
EG1005	ROUTE_POINT_EVENT	1005.857
EG1005	DROP_BOMB	1005.859
Bravo2	BATTLE_DAMAGE	1005.885
Bravo1	BATTLE_DAMAGE	1005.887
EG1015	ROUTE_POINT_EVENT	1006.483
EG1015	ROUTE_POINT_EVENT	1009.446
Bravo1	ROUTE_POINT_EVENT	1009.870
Bravo1	FIRE_CANNON	1009.872
EG1005	ROUTE_POINT_EVENT	1010.908
Bravo2	START_ENGINE_EVENT	1012.000
EG1015	ROUTE_POINT_EVENT	1012.767
Bravo3	START_ENGINE_EVENT	1013.000
Bravo3	ROUTE_POINT_EVENT	1013.619
Bravo2	ROUTE_POINT_EVENT	1013.763
Bravo1	ROUTE_POINT_EVENT	1016.271
EG1015	ROUTE_POINT_EVENT	1016.613
Bravo3	**BATTLE_DAMAGE**	**1017.360**
EG1005	ROUTE_POINT_EVENT	1018.133
EG1005	DROP_BOMB	1018.135
Bravo3	BATTLE_DAMAGE	1018.160
Bravo3	ROUTE_POINT_EVENT	1019.030
Bravo2	ROUTE_POINT_EVENT	1019.718
Bravo2	FIRE_CANNON	1019.720
EG1015	ROUTE_POINT_EVENT	1020.854
EG1005	ROUTE_POINT_EVENT	1021.808
EG1005	DROP_BOMB	1021.810
Bravo1	ROUTE_POINT_EVENT	1023.169
Bravo3	ROUTE_POINT_EVENT	1024.935
Bravo3	FIRE_CANNON	1024.937
EG1015	ROUTE_POINT_EVENT	1025.729
EG1005	ROUTE_POINT_EVENT	1025.730
Bravo2	ROUTE_POINT_EVENT	1025.841
Bravo2	BATTLE_DAMAGE	1029.878
Bravo1	ROUTE_POINT_EVENT	1030.305
EG1015	ROUTE_POINT_EVENT	1030.351
EG1005	ROUTE_POINT_EVENT	1030.352
EG1015	DROP_BOMB	1030.353
EG1005	DROP_BOMB	1030.354
Bravo2	BATTLE_DAMAGE	1030.378
Bravo3	ROUTE_POINT_EVENT	1031.335
Bravo2	BATTLE_DAMAGE	1031.878
Bravo2	ROUTE_POINT_EVENT	1032.387
EG1015	LANDED	1035.464

Node 1

**********PASE Simulation Results**********

Player's ID	Event Type	New Simulation Time
Bravo4	START_ENGINE_EVENT	999.000
Bravo4	ROUTE_POINT_EVENT	999.619
EG2025	TAKE_OFF_EVENT	1001.200
EG2025	ROUTE_POINT_EVENT	1001.973
EG2005	TAKE_OFF_EVENT	1002.500
EG2025	ROUTE_POINT_EVENT	1003.513
EG2025	**DROP_BOMB**	**1003.515**
EG2005	ROUTE_POINT_EVENT	1003.859
EG2005	DROP_BOMB	1003.861
EG2015	TAKE_OFF_EVENT	1004.500
Bravo4	ROUTE_POINT_EVENT	1005.030
EG2025	ROUTE_POINT_EVENT	1005.058
EG2025	**DROP_BOMB**	**1005.060**
EG2015	ROUTE_POINT_EVENT	1005.859
EG2015	DROP_BOMB	1005.861
EG2005	ROUTE_POINT_EVENT	1005.983
EG2015	ROUTE_POINT_EVENT	1007.983
EG2005	ROUTE_POINT_EVENT	1008.946
EG2025	ROUTE_POINT_EVENT	1010.108
Bravo4	ROUTE_POINT_EVENT	1010.935
Bravo4	FIRE_CANNON	1010.937
EG2015	ROUTE_POINT_EVENT	1010.946
EG2005	ROUTE_POINT_EVENT	1012.267
EG2015	ROUTE_POINT_EVENT	1014.267
Bravo5	START_ENGINE_EVENT	1015.000
Bravo5	ROUTE_POINT_EVENT	1015.619
EG2005	ROUTE_POINT_EVENT	1016.113
EG2025	ROUTE_POINT_EVENT	1017.333
EG2025	**DROP_BOMB**	**1017.335**
Bravo4	ROUTE_POINT_EVENT	1017.335
EG2015	ROUTE_POINT_EVENT	1018.113
EG2005	ROUTE_POINT_EVENT	1020.354
EG2025	ROUTE_POINT_EVENT	1021.008
EG2025	**DROP_BOMB**	**1021.010**
Bravo5	ROUTE_POINT_EVENT	1021.030
Bravo4	**BATTLE_DAMAGE**	**1021.035**
Bravo4	BATTLE_DAMAGE	1021.835
EG2015	ROUTE_POINT_EVENT	1022.354
Bravo4	ROUTE_POINT_EVENT	1024.207
EG2025	ROUTE_POINT_EVENT	1024.931
EG2005	ROUTE_POINT_EVENT	1025.229
Bravo5	ROUTE_POINT_EVENT	1026.870
Bravo5	FIRE_CANNON	1026.872
EG2015	ROUTE_POINT_EVENT	1027.229
EG2025	ROUTE_POINT_EVENT	1029.552
EG2025	**DROP_BOMB**	**1029.554**
Bravo5	**BATTLE_DAMAGE**	**1029.579**
EG2005	ROUTE_POINT_EVENT	1029.851
EG2005	DROP_BOMB	1029.853
Bravo5	BATTLE_DAMAGE	1030.379
Bravo4	ROUTE_POINT_EVENT	1031.846
EG2015	ROUTE_POINT_EVENT	1031.851
EG2015	DROP_BOMB	1031.853
Bravo5	ROUTE_POINT_EVENT	1033.271
EG2005	LANDED	1034.964
EG2025	LANDED	1035.100
EG2015	LANDED	1036.964
Bravo5	ROUTE_POINT_EVENT	1040.169

Node 0 (cont)

EG1005	LANDED	1035.899
Bravo1	ROUTE_POINT_EVENT	1037.731
Bravo3	ROUTE_POINT_EVENT	1038.207
Bravo2	ROUTE_POINT_EVENT	1039.616
Bravo3	ROUTE_POINT_EVENT	1045.846
Bravo1	ROUTE_POINT_EVENT	1045.938
Bravo1	FIRE_CANNON	1045.940
Bravo2	ROUTE_POINT_EVENT	1047.317
Bravo3	ROUTE_POINT_EVENT	1054.254
Bravo1	STOP_ENGINE	1054.788
Bravo2	ROUTE_POINT_EVENT	1055.740
Bravo2	FIRE_CANNON	1055.742
Bravo3	ROUTE_POINT_EVENT	1063.697
Bravo2	ROUTE_POINT_EVENT	1064.605
Bravo3	STOP_ENGINE	1073.364
Bravo2	STOP_ENGINE	1073.725
All_Players	**SIMULATION_END_EVENT**	**1100.000**

Node 1 (cont)

Bravo4	ROUTE_POINT_EVENT	1040.254
Bravo5	ROUTE_POINT_EVENT	1047.305
Bravo4	ROUTE_POINT_EVENT	1049.697
Bravo5	ROUTE_POINT_EVENT	1054.73
Bravo4	STOP_ENGINE	1059.364
Bravo5	ROUTE_POINT_EVENT	1062.938
Bravo5	FIRE_CANNON	1062.940
Bravo5	STOP_ENGINE	1071.788
All_Players	**SIMULATION_END_EVENT**	**1100.000**

Appendix C. PASS Code Templates

C.1 Introduction.

The following pages contain sample code templates for the Parallel Ada Simulation System (PASS). One template is provided for the Parallel Ada Simulation Model (PASM), and one for the Parallel Ada Simulation Environment (PASE). The code templates are explained in Chapter 4 Design/Implementation. The PASM sample is a template for a Player Class object. The PASE template is for an instance of the Protocol Filter Class (ie, Optimistic).

C.2 PASM Player Class Template.

```
--*****************************************************************************  *
--* File name:  xxxxxxxxx.xxx                                                    *
--*                                                                              *
--* Purpose:  This is the body for a Player Class Template.                     *
--*           Values in bold italics are "soft coded values                      *
--*            and should be modified to create a new                            *
--*            Player Class.                                                     *
--*                                                                              *
--* SubClasses: None.                                                            *
--*                                                                              *
--* Implemented by:  Author's Name                                               *
--*                                                                              *
--* Implementation Date: xx XXX xx.                                              *
--*                                                                              *
--* Modification Dates: No modifications.                                        *
--*                                                                              *
--* Language : Classic Ada PDL.                                                  *
--*                                                                              *
--* Compiler Dependencies:  Does not compile using the Meridian                  *
--*                                 Ada compiler.                                *
--*******************************************************************************
--
CLASS BODY Player_Class IS

    Instance_Variable : INSTANCE Instance_Variable_Type;
    -- All instance variables should be defined here

-- ***********************************************************************************************
-- * Create a new object.  This method may be called to create a new                             *
-- * Player_Class object with the following command:                                             *
-- *                                                                                             *
-- *              Send (Player_Class.Class_Object,                                               *
-- *                    Create,                                                                  *
-- *                    New_Object => New_Player_Class_Instance);                                *
-- ***********************************************************************************************
METHOD Create (New_Object : OUT Object_Id) is

  New_Instance : Object_Id := Instantiate;

begin
  SEND (New_Instance, Initialize);
  New_Object := New_Instance;
end Create;
```

```
--*****************************************************************************
--* Initializes each Player Class object                                      *
--*****************************************************************************
INSTANCE METHOD Initialize is
begin
   <sequence of statements>;
end Initialize;

--*****************************************************************************
--* Start, starts each Player Class object if it is implemented as a task.    *
--*****************************************************************************
INSTANCE METHOD Start is
begin
   null;
end Start;

--*****************************************************************************
--*  Set_Parameters initializes Player Class's parameters if any.             *
--*****************************************************************************
INSTANCE METHOD Set_Parameters(Data_File_Name : IN
                                  Simulation Types.Data_File_Name_Type) is

   Parameter_File     : Text_IO.File_Type;
   Parameter_Index   :  Integer;

begin
   Text_IO.OPEN  (File=>Parameter_File, Mode=>Text_IO.In_File,
                     Name=>Data_File_Name);
   Text_IO.Get_Line(File=>Parameter_File, Item=> <parameter>);
   For Parameter_Index in 1..Number_Of_Parameters loop
   Text_IO.Get_Line(File=>Parameter_File, Item => <Load_Next_Parameter>);
    -- Assign Local Value := Parameter;
   end loop;
   Text_IO.Close(File=>Parameter_File, Mode=>Text_IO.In_File,
                    Name=>Data_File_Name);
end Set_Parameters;

--*****************************************************************************
--* Set Parameter may be called to set the value of any parameter             *
--*****************************************************************************
INSTANCE METHOD Set (Parameter : IN Simulation_Types.Parameter_Type) is
 begin
   Current_Parameter := Parameter;
end Set;
```

```
--*********************************************************************************
--* Query Parameter can be used to return the value of an parameter      *
--*********************************************************************************
INSTANCE METHOD Query (Parameter : OUT
                              Simulation_Types.Parameter_Type) is
begin
   Current_Parameter := Parameter;
end Query;

--*********************************************************************************************
--* Process_Event processes events received from the Logical Process Class.      *
--*********************************************************************************************
INSTANCE METHOD Process_Event ( Event_To_Process : in Object_ID) is
   My_Event : Object_ID;
   -- identifies the next event to be processed

begin
   My_Event := Event_To_Process;
   SEND(My_Event, Get_Event_Type, E_Type => Type_Event);
   case Type_Event is

      when Event_1 =>
            <sequence of statements>

      when Event_2 =>
            <sequence of statements>
                  .
                  .
                  .
      when Event_N =>
            <sequence of statements>

      end case;

--*********************************************************************************
--* Generate_Event takes future events and passes them to the Logical  *
--*  Process  Class via the PASE_Interface to be placed on the NEQ.      *
--*********************************************************************************
INSTANCE METHOD Generate_Event ( Future_Event : in Object_ID) is
begin
   SEND(PASE_Interface, Schedule_Event, Event_To_Schedule =>
         Future_Event);
end Generate_Event;
end Player_Class;
```

C.3 PASE Protocol_Filter Class Template.

```
--*****************************************************************************  *
--* File name:  xxxxxxxxx.xxx                                                    *
--*                                                                              *
--* Purpose:  This is the body for a Filter Class Template.                      *
--*           Values in bold italics are "soft coded values                      *
--*           and should be modified to create a new                             *
--*           Player Class.                                                      *
--*                                                                              *
--* SubClasses: None.                                                            *
--*                                                                              *
--* Implemented by:  Author's Name                                               *
--*                                                                              *
--* Implementation Date: xx XXX xx.                                              *
--*                                                                              *
--* Modification Dates: No modifications.                                        *
--*                                                                              *
--* Language : Classic Ada PDL.                                                  *
--*                                                                              *
--* Compiler Dependencies:  Does not compile using the Meridian                  *
--*                         Ada compiler.                                        *
--*****************************************************************************
--
CLASS BODY Filter_Class IS

    Instance_Variable : INSTANCE Instance_Variable_Type;
    -- All instance variables should be defined here

-- ***************************************************************************************
-- * Create a new object.  This method may be called to create a new                     *
-- * Filter_Class object with the following command:                                     *
-- *                                                                                     *
-- *              Send (Filter_Class.Class_Object,                                       *
-- *                  Create,                                                            *
-- *                  New_Object => New_Filter_Class_Instance);                          *
-- ***************************************************************************************
METHOD Create (New_Object : OUT Object_Id) is

   New_Instance : Object_Id := Instantiate;

begin
   SEND (New_Instance, Initialize);
   New_Object := New_Instance;
end Create;
```

```
-- ********************************************************************************************  *
-- * Initialize the Filter class object.  This method performs all the necesary                 *
-- * operations required to initialize the Filter Class object.                                 *
-- ********************************************************************************************  *

INSTANCE METHOD Initialize is

begin
   <sequence of statements>;
end Initialize;

-- ********************************************************************************************  *
-- * Delete the Filter class object.  This method performs all the necesary                     *
-- * operations required to delete the Filter instance.                                         *
-- ********************************************************************************************  *

INSTANCE METHOD Delete is

begin
   SEND (SELF, Finalize);
   DESTROY;
end Delete;

-- ********************************************************************************************  *
-- * Finalize the Filter class object.  This method performs all the necesary                   *
-- * shutdown operations to instance variables                                                  *
-- ********************************************************************************************  *

INSTANCE METHOD Finalize is

begin
   null;
end Finalize;

-- ********************************************************************************  *****  *
-- * Safe_To_Proceed.  This method determines the safetime based on the                       *
-- * current implementation of the type of Filter being implemented,                          *
-- * and compares the proposed Time to the safetime.                                          *
-- *         If safetime > proposed time then safe := true,                                   *
-- *         Otherwise, safe := false.                                                        *
-- ********************************************************************************************
 INSTANCE METHOD Safe_To_Proceed ( Time   : in
                                   Simulation_Types.Simulation_Time_Type
                                   Safe   : out boolean)
```

```
    Proposed_Time :  Simulation_Types.Simulation_Time_Type;

  begin
    Proposed_Time := Time;
    <sequence of statements>
    -- insert statements here to implement desired protocol
    -- method to determine the safetime

    IF Proposed_Time > safetime THEN
        Safe := True;
    ELSE
        Safe := False;

  end Safe_To_Proceed;
end Filter_Class;
```

Appendix D. PASS Configuration Guide

D.1. Introduction.

The software required to run PASS can be broken into two categories: Support software and PASS software. This appendix lists the files required to execute PASS on a given host and provides the steps required to generate the PASS executables. Section D.2 lists the files and describes the procedure required to build the PASS for execution as a sequential discrete event simulation on the Sun Sparc stations or Intel Ipsc/2 Hypercube, and Section D.3 describes the generation of the PASS as a parallel discrete event simulation (PDES) for execution on the Hypercube.

D.2 Sequential Version of PASS.

D.2.1 Sequential Support Files.

The following support files are required to execute PASS.

- ***APP_TYPE.a*** Describes the types specific to a given application.
- ***ca_nxt_ins_o.a*** Classic Ada support file.
- ***classic_ex_s.a*** Classic Ada support file.
- ***classic_typ.a*** Classic Ada support file.
- ***Event_IO_Pkg.a*** Instantiation of the generic Enumeration_IO package.
- ***file_cont_b.a*** Body for file control package.
- ***file_cont_s.a*** Specification for file control package.
- ***list_b.a*** Body for Linked List package.
- ***list_s.a*** Specification for Linked List package.
- ***lp_hash.a*** Instantiation of generic Map package.
- ***lst_srch_b.a*** Body for list search package.
- ***lst_srch_s.a*** Specification for list search package.

- ***lst_utl_b.a*** Body for list utilities package.
- ***lst_utl_s.a*** Specification for list utilities package.
- ***map_b.a*** Body for generic map package.
- ***map_s.a*** Specification for generic map package.
- ***npom_body.a*** Classic Ada support package.
- ***npom_spec.a*** Classic Ada Support package.
- ***ObjLstP.a*** Instantiation of generic list package.
- ***p_obj_base_b.a*** Classic Ada Support package.
- ***p_obj_base_s.a*** Classic Ada Support package.
- ***pom_body.a*** Classic Ada Support package.
- ***pom_spec.a*** Classic Ada Support package.
- ***SimTimeIoP.a*** Instantiation of generic Fixed_IO package.
- ***SIM_TYPES.a*** Definition of PASS simulation types.
- ***SmFltIoP.a*** Instantiation of generic Float_IO package.
- ***SStorManB.a*** Body for storage manager utilities package.
- ***SStorManS.a*** Specification for storage manager utilities package.
- ***stat_str_b.a*** Body for generic static string operations package.
- ***stat_str_s.a*** Specification for generic static string operations package.
- ***strg_op.a*** Instantiation of generic static string operations package.
- ***adaorder*** Generates a Meridian Ada compilation order file.
- ***quietorder*** Converts Meridian Ada order file to a Verdix Ada compilation file.
- ***order*** Compilation order for the *support files.*

D.2.2 Sequential PASS Files.

The files required to create PASS are the .CA (Classic Ada) files listed in the following make file named *build-pase* (# indicates a comment):

```
# Builds the PASE simulation
setenv CA_ERROR_LIM 20
setenv CA_SCRIPT_PREFIX csh
ca.version
rm -r ../classlib
mkdir ../classlib
ca.create -d ../classlib
ca.set -d ../classlib
rm -r ../adalib
mkdir ../adalib
cd ../adalib
a.mklib -f
a.path -a ~jbelford/CUBE_PASE/support-packages
ln -s ~jbelford/Sequential_PASE base_classes
# processing class specifications of PASE basis simulation classes
echo processing class specifications of PASE basis simulation classes
echo classic base_classes/PASE_INT_S.CA
classic base_classes/PASE_INT_S.CA
echo classic base_classes/ROOT_EVENT_S.CA
classic base_classes/ROOT_EVENT_S.CA
echo classic base_classes/AC_EVENT_S.CA
classic base_classes/AC_EVENT_S.CA
echo classic base_classes/TANK_EVENT_S.CA
classic base_classes/TANK_EVENT_S.CA
echo classic base_classes/LOCATION_S.CA
classic base_classes/LOCATION_S.CA
echo classic base_classes/ROUTE_PNT_S.CA
classic base_classes/ROUTE_PNT_S.CA
echo classic base_classes/ROUTE_S.CA
classic base_classes/ROUTE_S.CA
echo classic base_classes/PLAYER_S.CA
classic base_classes/PLAYER_S.CA
echo classic base_classes/VEHICLE_S.CA
classic base_classes/VEHICLE_S.CA
echo classic base_classes/AIRCRAFT_S.CA
classic base_classes/AIRCRAFT_S.CA
echo classic base_classes/TANK_S.CA
classic base_classes/TANK_S.CA
echo classic base_classes/IO_MGR_S.CA
classic base_classes/IO_MGR_S.CA
echo classic base_classes/FILE_S.CA
classic base_classes/FILE_S.CA
echo classic base_classes/MONITOR_S.CA
classic base_classes/MONITOR_S.CA
echo classic base_classes/SYNCHRO_S.CA
classic base_classes/SYNCHRO_S.CA
echo classic base_classes/CLOCK_S.CA
classic base_classes/CLOCK_S.CA
echo classic base_classes/NEQ_S.CA
classic base_classes/NEQ_S.CA
echo classic base_classes/FILTER_S.CA
classic base_classes/FILTER_S.CA
```

```
echo classic base_classes/CONSERV_S.CA
classic base_classes/CONSERV_S.CA
echo classic base_classes/LP_MGR_S.CA
classic base_classes/LP_MGR_S.CA
echo classic base_classes/NODE_MGR_S.CA
classic base_classes/NODE_MGR_S.CA
echo classic base_classes/HYPERCUBE_S.CA
classic base_classes/HYPERCUBE_S.CA
# processing class bodies of PASE basis simulation classes
echo processing class bodies of PASE basis simulation classes
echo classic base_classes/PASE_INT_B.CA
classic base_classes/PASE_INT_B.CA
echo classic base_classes/ROOT_EVENT_B.CA
classic base_classes/ROOT_EVENT_B.CA
echo classic base_classes/AC_EVENT_B.CA
classic base_classes/AC_EVENT_B.CA
echo classic base_classes/TANK_EVENT_B.CA
classic base_classes/TANK_EVENT_B.CA
echo classic base_classes/LOCATION_B.CA
classic base_classes/LOCATION_B.CA
echo classic base_classes/ROUTE_PNT_B.CA
classic base_classes/ROUTE_PNT_B.CA
echo classic base_classes/ROUTE_B.CA
classic base_classes/ROUTE_B.CA
echo classic base_classes/PLAYER_B.CA
classic base_classes/PLAYER_B.CA
echo classic base_classes/VEHICLE_B.CA
classic base_classes/VEHICLE_B.CA
echo classic base_classes/AIRCRAFT_B.CA
classic base_classes/AIRCRAFT_B.CA
echo classic base_classes/TANK_B.CA
classic base_classes/TANK_B.CA
echo classic base_classes/IO_MGR_B.CA
classic base_classes/IO_MGR_B.CA
echo classic base_classes/FILE_B.CA
classic base_classes/FILE_B.CA
echo classic base_classes/MONITOR_B.CA
classic base_classes/MONITOR_B.CA
echo classic base_classes/SYNCHRO_B.CA
classic base_classes/SYNCHRO_B.CA
echo classic base_classes/CLOCK_B.CA
classic base_classes/CLOCK_B.CA
echo classic base_classes/NEQ_B.CA
classic base_classes/NEQ_B.CA
echo classic base_classes/FILTER_B.CA
classic base_classes/FILTER_B.CA
echo classic base_classes/CONSERV_B.CA
classic base_classes/CONSERV_B.CA
echo classic base_classes/LP_MGR_B.CA
classic base_classes/LP_MGR_B.CA
echo classic base_classes/NODE_MGR_B.CA
classic base_classes/NODE_MGR_B.CA
```

```
classic base_classes/HYPERCUBE_B.CA
ca.generate
~jbelford/utilities/adaorder *.a
~jbelford/utilities/quietorder
order
rm ~jbelford/adalib/*.sh
# echo Processing the PASE main program
classic base_classes/PASE.CA
ada PASE.a
echo Loading and Linking PASE
a.ld PASE -o SEQ-PASE
```

D.2.3 Building Sequential PASS.

The Sequential version of PASS can be created as follows:

Step 1: Create a support directory and ada library then copy all of the support files listed in Section D.2.1 into that directory (use the order file to compile them).

Step 2: Create a PASS directory and copy all of the PASS .CA files listed in the build file provided in Section D.2.2 into that directory.

Step 3: Modify the build file (*build-pase*) to provide visibility into the new directories and copy it into the same directory as the PASS code.

Step 4: Execute the *build-pase* file.

Steps 5 - 6 are required to port PASS to the Hypercube.

Step 5: Change to the directory containing the ada code (adalib).

Step 6: Edit the file cae_1_body.a.

1) Change the *case* discrete expression *class_id* to *abs(class_id)*.

2) Change all of the negative *case* values to positive integers (e.g., -10 to 10).

note: Classic Ada version 9.0 is the only version which requires this. Later versions do not generate negative *case* values.

Step 7: Rename file classic_executive_body.a to classic_ex_b.a.

Step 8: Copy all .a files, support files, and the order files to the cube.

Step 9: Execute the order files to compile the ada files and the support files.

Step 10: Create an executable by typing

a.ld PASE1 -o SEQ-PASE -lnode.

D.3 Parallel Version of PASS.

D.3.1 Parallel Support Files.

The same support files required for the sequential version of PASS that are listed in Section D.2.1, are also required for the parallel version.

D.3.2 Parallel PASS Files.

The files required to create the parallel version of the PASS are the .CA (Classic Ada) files listed in the following make file named *create-cube-code* (# indicates a comment):

```
# creates the ada code for the cube
setenv CA_ERROR_LIM 20
setenv CA_SCRIPT_PREFIX csh
ca.version
rm -r ../classlib
mkdir ../classlib
ca.create -d ../classlib
ca.set -d ../classlib
rm -r ../adalib
mkdir ../adalib
cd ../adalib
a.mklib -f
# provide visibility to support packages
a.path -a ~jbelford/CUBE_PASE/support-packages
ln -s ~jbelford/CUBE_PASE/CUBE_CODE base_classes
# processing class specifications of PASE basis simulation classes
echo processing class specifications of PASE basis simulation classes
echo classic base_classes/PASE_INT_S.CA
classic base_classes/PASE_INT_S.CA
echo classic base_classes/ROOT_EVENT_S.CA
classic base_classes/ROOT_EVENT_S.CA
echo classic base_classes/AC_EVENT_S.CA
classic base_classes/AC_EVENT_S.CA
echo classic base_classes/TANK_EVENT_S.CA
classic base_classes/TANK_EVENT_S.CA
echo classic base_classes/LOCATION_S.CA
classic base_classes/LOCATION_S.CA
echo classic base_classes/ROUTE_PNT_S.CA
classic base_classes/ROUTE_PNT_S.CA
echo classic base_classes/ROUTE_S.CA
classic base_classes/ROUTE_S.CA
echo classic base_classes/PLAYER_S.CA
classic base_classes/PLAYER_S.CA
echo classic base_classes/VEHICLE_S.CA
classic base_classes/VEHICLE_S.CA
echo classic base_classes/AIRCRAFT_S.CA
classic base_classes/AIRCRAFT_S.CA
```

```
echo classic base_classes/TANK_S.CA
classic base_classes/TANK_S.CA
echo classic base_classes/IO_MGR_S.CA
classic base_classes/IO_MGR_S.CA
echo classic base_classes/FILE_S.CA
classic base_classes/FILE_S.CA
echo classic base_classes/MONITOR_S.CA
classic base_classes/MONITOR_S.CA
echo classic base_classes/SYNCHRO_S.CA
classic base_classes/SYNCHRO_S.CA
echo classic base_classes/CLOCK_S.CA
classic base_classes/CLOCK_S.CA
echo classic base_classes/NEQ_S.CA
classic base_classes/NEQ_S.CA
echo classic base_classes/FILTER_S.CA
classic base_classes/FILTER_S.CA
echo classic base_classes/CONSERV_S.CA
classic base_classes/CONSERV_S.CA
echo classic base_classes/LP_MGR_S.CA
classic base_classes/LP_MGR_S.CA
echo classic base_classes/NODE_MGR_S.CA
classic base_classes/NODE_MGR_S.CA
echo classic base_classes/REM_NODE_S.CA
classic base_classes/REM_NODE_S.CA
echo classic base_classes/HYPERCUBE_S.CA
classic base_classes/HYPERCUBE_S.CA
# processing class bodies of PASE basis simulation classes
echo processing class bodies of PASE basis simulation classes
echo classic base_classes/PASE_INT_B.CA
classic base_classes/PASE_INT_B.CA
echo classic base_classes/ROOT_EVENT_B.CA
classic base_classes/ROOT_EVENT_B.CA
echo classic base_classes/AC_EVENT_B.CA
classic base_classes/AC_EVENT_B.CA
echo classic base_classes/TANK_EVENT_B.CA
classic base_classes/TANK_EVENT_B.CA
echo classic base_classes/LOCATION_B.CA
classic base_classes/LOCATION_B.CA
echo classic base_classes/ROUTE_PNT_B.CA
classic base_classes/ROUTE_PNT_B.CA
echo classic base_classes/ROUTE_B.CA
classic base_classes/ROUTE_B.CA
echo classic base_classes/PLAYER_B.CA
classic base_classes/PLAYER_B.CA
echo classic base_classes/VEHICLE_B.CA
classic base_classes/VEHICLE_B.CA
echo classic base_classes/AIRCRAFT_B.CA
classic base_classes/AIRCRAFT_B.CA
echo classic base_classes/TANK_B.CA
classic base_classes/TANK_B.CA
echo classic base_classes/IO_MGR_B.CA
classic base_classes/IO_MGR_B.CA
```

```
echo classic base_classes/FILE_B.CA
classic base_classes/FILE_B.CA
echo classic base_classes/MONITOR_B.CA
classic base_classes/MONITOR_B.CA
echo classic base_classes/SYNCHRO_B.CA
classic base_classes/SYNCHRO_B.CA
echo classic base_classes/CLOCK_B.CA
classic base_classes/CLOCK_B.CA
echo classic base_classes/NEQ_B.CA
classic base_classes/NEQ_B.CA
echo classic base_classes/FILTER_B.CA
classic base_classes/FILTER_B.CA
echo classic base_classes/CONSERV_B.CA
classic base_classes/CONSERV_B.CA
echo classic base_classes/LP_MGR_B.CA
classic base_classes/LP_MGR_B.CA
echo classic base_classes/NODE_MGR_B.CA
classic base_classes/NODE_MGR_B.CA
echo classic base_classes/REM_NODE_B.CA
classic base_classes/REM_NODE_B.CA
echo classic base_classes/HYPERCUBE_B.CA
classic base_classes/HYPERCUBE_B.CA
# ca.generate creates the classic ada bodies for the code
ca.generate
echo classic base_classes/PASE_MAIN.CA
classic base_classes/PASE_MAIN.CA
# creates the ada order file for compiling the code.
~jbelford/utilities/adaorder *.a
~jbelford/utilities/quietorder
rm ~jbelford/CUBE_PASE/adalib/*.sh
```

The files *REM_NODE_S.CA* and *REM_NODE_B.CA* have been added for the parallel version of PASS. In addition, the code in the files *LP_MGR_B.CA, SYNCHRO_B.CA, PASE_MAIN_B.CA, NODE_MGR_B.CA, and HYPERCUBE_B.CA* is different from the files with the same name for the sequential version.

D.3.3 Building Parallel PASS.

The steps for building the parallel version of the PASS are the same as those listed in Section D.2.3 for the sequential version. However, the .CA files should be copied from the CUBE_CODE directory. Also, all of the optional steps required for the Hypercube must be followed. Step 10 would now be

Create an executable by typing

```
a.ld PASE1 -o MAIN -lnode.
```

Appendix E provides the instructions for executing a simulation.

Appendix E. PASS Software User's Guide

E.1 Introduction.

This appendix briefly describes how to execute a simulation using the Parallel Ada Simulation System (PASS).

E.2 Executing a PASS Simulation.

E.2.1 Sequential Mode.

To execute a PASS simulation in the sequential mode the following files should be in placed in a single directory:

1) The executable generated according to Section D.2.3 (*SEQ-PASE*).

2) The applicable Route files.

3) The application file (*application.inf*).

After setting up the application file according to the directions within the file, simply execute the file *SEQ-PASE*. The results will be written to a file named *simulation.dat*.

On the Hypercube execute the following commands:

> getcube -t d0

> load SEQ-PASE

E.2.2 Parallel Mode.

To execute a PASS simulation in the parallel mode the following files should be in placed in a single directory on the cube:

1) The executable generated according to Section D.3.3 (*MAIN*).

2) The applicable Route files.

3) An application file for each node in the simulation (i.e., *application.0 - application.7*).

Execute the following commands:

> getcube -t d*n* (where *n* represents the dimension of the desired cube)

> load MAIN

Bibliography

1. Barnes, John. "Ada 9X Project Report". Reading, England. February 1993.

2. Booch, Grady. Software Components with Ada: *Structures, Tools, and Subsystems.* The Benjamin/ Cummins Publishing Company, Inc., 1987.

3. Booch, Grady. Software Engineering with Ada. (Second Edition). The Benjamin/ Cummins Publishing Company, Inc., 1986.

4. Brown Randy E. and William K. McQuay, The Joint Modeling and Simulation System. Journal of Electronic Defense, September 1992.

5. Chandy, K. M. and J. Misra. Asynchronous Distributed Simulation via a Sequence of Parallel Computations. Communications of the ACM, 24(11):198-206 (May 1981).

6. DeCegama, Angel L. The Technology of Parallel Processing. New Jersey: Prentice Hall, 1989.

7. Fujimoto, Richard M. Parallel Discrete-Event Simulation. NSF grants DCR-8504826 and CCR-8902362. School of Information and Computer Science, Georgia Institute of Technology, Atlanta GA.

8. Hartrum, Thomas C. "AFIT Guide to SPECTRUM." October 1992, Unpublished Report.

9. Hartrum, Thomas C. "Project Q: A Case Study In Parallel Discrete Event Simulation." May 1992, Unpublished Report.

10. Hartrum, Thomas C. TCHSIM: A Simulation Environment for Parallel Discrete Event Simulation. AFIT/ENG. October 1992, Unpublished Report.

11. Institute for Simulation and Training. Protocol Data Units for Entity Information and Entity Interaction in a Distributed Interactive Simulation. Military Standard, version 2.0 (draft). September 4, 1992.

12. J-MASS Architectural Working Group. Software Development Plan (SDP) Volume II, Software Structural Model, Design Methodology for the Modeling Library Components for the J-MASS Program. Ver. 2.0. ASD/RWWW, WPAFB, Ohio. February 26, 1993.

13. Misra, Jayadev. Distributed Discrete-Event Simulation. Computing Surveys, Vol 18, No. 1, March 1986.

14. Presscott, John Van Horn. Development of a Protocol Usage Guideline for Conservative Parallel Simulations. MS thesis, AFIT/GCS/ENG/GCS92D-19. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1992.

15. Reynolds, Jr., Paul F. "A Spectrum of Options for Parallel Simulation." Proceedings of the ACM Winter Simulation Conference. 1988.

16. Rumbaugh, James, et al. Object-Oriented Modeling and Design. Englewood Cliffs, New Jersey: Prentice-Hall, Inc, 1991.

17. Schuppe, Thomas F. "Modeling and Simulation: A Department of Defense Critical Technology." Proceedings of the 1991 Winter Simulation Conference. 519 - 525. San Diego: The Society for Computer Simulation International, 1991.

18. Software Productivity Solutions, Inc. Classic Ada - User's Manual. Ver 9.0, 1992.

19. Vaden, David W. Distributed Interactive Simulation... Vision for the Next Decade. Army Research Development and Acquisition Bulletin, May-June 1993.

Vita

Captain James T. Belford was born in Chicago, Illinois, on 12 October 1958. He entered the Air Force on 17 May 1976 as a jet engine technician. In May of 1983, he was selected for the Airmen's Education and Commissioning Program (AECP) and attended California State University in Sacramento, California. In June 1987 he was awarded a Bachelor of Science Degree in Electrical and Electronic Engineering. After getting his commission in the United States Air Force through the Officer Training School in October 1987, he was assigned to the Air Force Systems Command, Munitions System Division, at Eglin AFB in Florida. There he held the position of Lead Engineer for Scoring Systems until he was selected to attend the Air Force Institute of Technology in April 1992.

Permanent Address: 1868 E. 224th Street
Sauk Village, Illinois 60411

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	November 1993	Master's Thesis

4. TITLE AND SUBTITLE
Object-Oriented Design and Implementation of a Parallel Ada Simulation System

5. FUNDING NUMBERS

6. AUTHOR(S)
James T. Belford, Capt, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology, WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GCE/ENG/93D-01

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Capt Rick Painter
2241 Avionics Circle, Suite 16
WL/AAWA-1 BLD 620
Wright-Patterson AFB, OH 45433-7765

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Simulations which model the behavior of "real world" entities are often large and complex, and require frequent changes to the configuration. This research effort examines the benefits of using object-oriented techniques to develop a distributed simulation environment which supports modularity, modifiability, and portability.

The components of a Parallel Discrete Event Simulation (PDES) environment are identified and modeled using the Rumbaugh modeling technique. From the model, a prototype implementation of a Parallel Ada Simulation Environment (PASE) is accomplished using Classic Ada. A system interface for the Intel ipsc/2 Hypercube was developed to illustrate the concepts of modularity and portability. In addition, the prototype environment uses a filter which implements the basic Chandy-Misra time synchronization protocol.

Finally, To test the correct operation of the environment, a simple battlefield application model is developed. PASE is tested in the sequential mode on both a Sun Sparc station and the Hypercube. The ability to distribute across *n*-nodes is demonstrated using various configurations on the Hypercube. The parallel test demonstrates the ability for objects on separate processors to interact with each other by passing messages, and to execute events generated by remote objects in the proper time stamp order.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Parallel Simulation, Distributed Simulation, Discrete Event Simulation, Object-Oriented	112
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

GENERAL INSTRUCTIONS FOR COMPLETING SF 298

The Report Documentation Page (RDP) is used in announcing and cataloging reports. It is important that this information be consistent with the rest of the report, particularly the cover and title page. Instructions for filling in each block of the form follow. It is important to ***stay within the lines*** to meet ***optical scanning requirements***.

Block 1. Agency Use Only *(Leave blank)*.

Block 2. Report Date. Full publication date including day, month, and year, if available (e.g. 1 Jan 88). Must cite at least the year.

Block 3. Type of Report and Dates Covered. State whether report is interim, final, etc. If applicable, enter inclusive report dates (e.g. 10 Jun 87 - 30 Jun 88).

Block 4. Title and Subtitle. A title is taken from the part of the report that provides the most meaningful and complete information. When a report is prepared in more than one volume, repeat the primary title, add volume number, and include subtitle for the specific volume. On classified documents enter the title classification in parentheses.

Block 5. Funding Numbers. To include contract and grant numbers; may include program element number(s), project number(s), task number(s), and work unit number(s). Use the following labels:

C	- Contract	**PR**	- Project
G	- Grant	**TA**	- Task
PE	- Program Element	**WU**	- Work Unit Accession No.

Block 6. Author(s). Name(s) of person(s) responsible for writing the report, performing the research, or credited with the content of the report. If editor or compiler, this should follow the name(s).

Block 7. Performing Organization Name(s) and Address(es). Self-explanatory.

Block 8. Performing Organization Report Number. Enter the unique alphanumeric report number(s) assigned by the organization performing the report.

Block 9. Sponsoring/Monitoring Agency Name(s) and Address(es). Self-explanatory.

Block 10. Sponsoring/Monitoring Agency Report Number. *(If known)*

Block 11. Supplementary Notes. Enter information not included elsewhere such as: Prepared in cooperation with...; Trans. of...; To be published in.... When a report is revised, include a statement whether the new report supersedes or supplements the older report.

Block 12a. Distribution/Availability Statement. Denotes public availability or limitations. Cite any availability to the public. Enter additional limitations or special markings in all capitals (e.g. NOFORN, REL, ITAR).

- **DOD** - See DoDD 5230.24, "Distribution Statements on Technical Documents."
- **DOE** - See authorities.
- **NASA** - See Handbook NHB 2200.2.
- **NTIS** - Leave blank.

Block 12b. Distribution Code.

- **DOD** - Leave blank.
- **DOE** - Enter DOE distribution categories from the Standard Distribution for Unclassified Scientific and Technical Reports.
- **NASA** - Leave blank.
- **NTIS** - Leave blank.

Block 13. Abstract. Include a brief *(Maximum 200 words)* factual summary of the most significant information contained in the report.

Block 14. Subject Terms. Keywords or phrases identifying major subjects in the report.

Block 15. Number of Pages. Enter the total number of pages.

Block 16. Price Code. Enter appropriate price code *(NTIS only)*.

Blocks 17. - 19. Security Classifications. Self-explanatory. Enter U.S. Security Classification in accordance with U.S. Security Regulations (i.e., UNCLASSIFIED). If form contains classified information, stamp classification on the top and bottom of the page.

Block 20. Limitation of Abstract. This block must be completed to assign a limitation to the abstract. Enter either UL (unlimited) or SAR (same as report). An entry in this block is necessary if the abstract is to be limited. If blank, the abstract is assumed to be unlimited.

★U.S.GPO:1990-0-273-071